University of Oxford
Computing Laboratory

MSc Computer Science Dissertation

Analysing Document Similarity Measures

Author:
Edward Grefenstette

Supervisor:
Professor Stephen Pulman

November 10, 2010

Abstract

The concept of a Document Similarity Measure is ill-defined due to the wide variety of existing metrics measuring a range of often very different notions of similarity. The crucial role of measuring document similarity in a wide variety of text and language processing tasks calls not only for a better understanding of how metrics work in general terms, but also a better way of analysing, comparing and improving document similarity measures.

This dissertation supplies an overview of different theoretical positions on the notion of similarity, discusses the construction of a framework for analysing and comparing a wide variety of metrics against a testbed corpus subdivided into categories by similarity type, and finally presents the results of an experiment run using this framework.

We found that the metric rankings and breakdown of results justified the theoretical position we developed while considering the nature of similarity, and that there is no deep distinction between syntactic, lexical and semantic similarity features. The results of our experiments also allowed us to posit various methods for developing better document similarity measures, using the output of the analysis framework to motivate their construction.

Contents

Abstract **3**

List of Figures **9**

List of Tables **11**

Acknowledgements **13**

1 Introduction **15**

2 Background and Theory **19**

2.1 Philosophical Foundations 19
2.1.1 Platonist Similarity 20
2.1.2 Similarity and the Foundation of Mathematics 20
2.1.3 Davidson on Similarity in Metaphors 21
2.1.4 Similarity in the Cognitive Sciences 22
2.1.5 Wittgenstein on Similarity 23
2.2 Different kinds of Document Similarity 25
2.2.1 Three Classes of Document Similarity 25
2.2.2 Similarity Features 26
2.2.3 Cross-Category Similarity Features 27
2.2.4 Classifying Similarity Types 28
2.3 Dropping the Class Distinction 30

3 Methodology **33**

3.1 Constructing a Corpus 34
3.1.1 Selecting Document Similarity Types 34
3.1.2 Designing a Corpus Construction Framework 37

3.1.3 Setting the Gold-Standard 38
3.1.4 Mixing Things Up (If Needed) 39
3.2 Evaluating Metrics: Considerations and Concerns 41
3.2.1 Dealing with the Corpus 42
3.2.2 Normalising the Results 43
3.2.3 Adapting Word/Character-based Metrics 43
3.3 Analysis: The Devil in the Details 44
3.3.1 Gold-Standard Ranking vs. Retrieval 45
3.3.2 Finer-grain Analysis: Breaking Down Results 47

4 Designing a Corpus Construction Module 51
4.1 Setting Up a General Framework 51
4.1.1 Structure and Shared Features 51
4.1.2 Suggested Improvements 55
4.2 Syntactic Toy-Corpora 55
4.2.1 General Overview 55
4.2.2 Construction Process: Random Edits Corpus 56
4.2.3 Construction Process: POS-Tag Equivalence Corpus . . 59
4.3 Theme-grouped Texts 61
4.3.1 General Overview 61
4.3.2 Construction Process 61
4.4 Wikipedia Article Pairs 63
4.4.1 General Overview 63
4.4.2 Background and Sources 63
4.4.3 Construction Process 64
4.4.4 Future Improvements 65
4.5 Paraphrase Corpora 66
4.5.1 General Overview 66
4.5.2 Background and Sources 66
4.5.3 Construction Process 67
4.5.4 Future Improvements 70
4.6 Abstract-Paper Pairs 70
4.6.1 General Overview 70
4.6.2 Background and Sources 71
4.6.3 Construction Process 71
4.6.4 Future Improvements 72

5 Evaluating Document Similarity Measures 75

5.1 Generalised Evaluation Framework . . . 76
5.2 Implementing Purely Syntactic Metrics . . . 77
5.2.1 Character-Count and Word-Count . . . 78
5.2.2 Levenshtein Edit Distance . . . 78
5.2.3 Jaro-Winkler Distance . . . 79
5.2.4 Ratio Similarity . . . 80
5.2.5 Jaccard Similarity . . . 80
5.2.6 MASI Distance . . . 81
5.3 BLEU: A Machine Translation Evaluation Metric . . . 82
5.3.1 Origin and use . . . 82
5.3.2 Implementation and score normalisation . . . 83
5.4 Wordnet-based Metrics . . . 83
5.4.1 Overview of Wordnet Metrics . . . 84
5.4.2 Implementing Wordnet-based Metrics . . . 86
5.4.3 Fine-tuning: How Far is Too Far? . . . 87
5.5 Semantic Vectors . . . 88
5.5.1 Background Theory . . . 89
5.5.2 Implementation: A Different Approach to Word-Sentence Scaling . . . 90

6 Results and Analysis 93
6.1 Structure of Analysis Script . . . 93
6.2 Discussion of Metric CGS Scores . . . 95
6.2.1 Overall Rankings . . . 95
6.2.2 Random Edits Analysis . . . 96
6.2.3 POS-Switch Analysis . . . 100
6.2.4 Theme Analysis . . . 101
6.2.5 Wikipedia Analysis . . . 102
6.2.6 Paraphrase Analysis . . . 103
6.2.7 Abstract-Article Analysis . . . 104

7 Conclusions 119

Bibliography 123

A Technical Details 127
A.1 Supplied CD-ROM Contents . . . 127
A.2 System Requirements . . . 130

B Source Code **131**

B.1 corpusbuilder.py 131
B.2 buildcorpus.py 140
B.3 prepcorpus.py 143
B.4 corpusevaluation.py 145
B.5 semanticvectors.java 152
B.6 evaluatecorpus.py 155
B.7 analyser.py 157

List of Figures

2.1 Classes, Properties and Types of Document Similarity 26

3.1 Example of Visually Represented CGS Score Distribution . . . 47

6.1 Levenshtein Edit Distance CGS Score Distribution (Edits Section) 98
6.2 Jaro-Winkler Distance CGS Score Distribution (Edits Section) 99
6.3 Comparison of BLEU and Path Similarity for POS-switch Docs 107
6.4 Comparison of BLEU and Path Similarity for Matched POS-switch Docs 108
6.5 Comparison of BLEU and Path Similarity for Mixed POS-switch Docs 109
6.6 Comparison of Character Count and MASI for Theme Documents 110
6.7 Breakdown of Semantic Vectors CGS Distribution for Wikipedia Docs 111
6.8 Comparison of Semantic Vectors and Path Sim. for Paraphrase Docs 112
6.9 Comparison of Semantic Vectors and Path Sim. for Matched Paraphrase Docs 113
6.10 Comparison of Semantic Vectors and Path Sim. for Mixed Paraphrase Docs 114
6.11 Comparison of Jaccard and MASI Distances for Paraphrase Docs 115
6.12 Comparison of Jaccard and MASI Distances for Matched Paraphrase Docs 116
6.13 Comparison of Jaccard and MASI Distances for Mixed Paraphrase Docs 117

6.14 Semantic Vectors CGS Score Distribution (Abstracts) 118

List of Tables

6.1 Corpus Category Legend 96
6.2 Overall Rankings (% CGS Scores) 97
6.3 Matched Document Pair Rankings (% CGS Scores) 97
6.4 Mixed Document Pair Rankings (% CGS Scores) 106

Acknolwedgements

First and foremost, thanks to Stephen Pulman for his supervision and his support throughout this project for which he proposed the subject matter in the first place. His constant feedback and bibliographical pointers were always helpful, and this work would not have been of the same quality if he had not taken time out of his packed head-of-department schedule to arrange regular supervision meetings.

Much gratefulness is due to Edward Loper and Stephen Bird, not only for developing the Natural Language Processing Toolkit and writing an amazingly clear companion book, but also for taking the time to answer my many questions about both the toolkit's technical hitches and how to approach specific issues of metric evaluation, which came in handy while thinking about how to deal with word-based Wordnet metrics.

Thanks also to Pascal Combescot for his helpful suggestions in the early stages of this project, and for sharing his implementation of Dominic Widdows' excellent Semantic Vectors package; to Simon Zwarts for his C implementation of the BLEU metric; to the anonymous coders behind the Poppler PDF tools and those behind the Levenshtein metric package; and to Sean Heelan for (unsuccessfully) helping me troubleshoot my installation of the METEOR metric for evaluating machine translation, which didn't make the cut. We tried.

Finally, thanks to my parents Irene and Greg Grefenstette, as well as my grandparents for their support of various sorts over the years, without which I wouldn't have had the peace of mind to finish large projects like this one on time.

Edward T. Grefenstette

Chapter 1

Introduction

Evaluating the similarity between two documents is an operation which lies at the heart of most text and language processing tasks. Once we consider 'document' to mean not a file or placeholder for information content as dictated by the everyday use of the term, but rather a delineable unit of information (be it a paragraph, an article, a sentence or even a word, in the case of textually represented information), then the fact that evaluating document similarity is an essential operation to such tasks should become intuitively clear by examination of a few common examples of text or language-processing tasks. We give systems which perform such evaluations the name of *document similarity measures* (DSMs) or document similarity metrics[1].

To give a few examples: when we make use of a search engine, we request web-page documents which bear *some similarity* to the keywords or string(s) which constitute the query document; when we ask for a text in some language A to be translated into some language B, we request a document in language B which has *some similarity* to the document in language A; when we summarise one document, we seek to produce another document which is *similar in some ways* (*e.g.* core meaning being preserved), which however is also different in other ways (the summary must be shorter than the original document, sentences are more compact). Naturally, such use of document similarity could easily involve other media than text, such as pictures, film,

[1] Both terms are generally used interchangeably in the literature, and will be used so here.

audio, etc. However, the metrics for such media can understandably be very different (and often more primitive) from the ones we will be considering in text processing, and therefore we will consider non-text documents to be outside of the scope of this discussion.

The above few examples illustrate an interesting point, which would be equally observed in any further examples of document similarity use: although the abstract task being completed in each example—that of attempting to compute *some sort of similarity* between sets of documents in order to achieve some practical goal (classification, validation, generation, etc.)—is the same in each case, the notion of similarity at play is not. Indeed, the similarity we draw upon when ranking documents according to word-count is much different than that which we use when translating sentences, which itself is very different from that which we use when summarising text. In the word-count case, we care only about superficial syntactic features; in the translation case, perhaps we look for shared lexical features, and in the summarisation case we not only draw upon semantic features, but also upon syntactic features similar to those exploited in the word-count case (since we want a shorter summary document).

The observation that DSMs are systems that perform the same abstract task while drawing upon very different aspects of documents depending on the comparison goals raises a few questions about the nature of DSMs. What is the common thread to DSM design? Is it a software engineering problem, or are there general principles underlying their construction? Are metrics designed for one purpose suitable for another? How would we determine this if they were? How do DSMs deal with different kinds of input (words, sentences, sets of paragraphs)? On what grounds can we compare metrics? How do we choose a 'better' metric relative to a task?

This jumble of questions justifies further investigation, but leaves us with little clue as to how to begin. Attempts have been made in the computational linguistics literature to answer some of these questions with regard to small groups of metrics, particularly within the context of comparing two specific types of metrics (*e.g.* see (Agirre et al. 2009)), however we found no attempt at giving a general theory of DSM design and analysis in the literature, and have resolved to approach the problem ourselves.

The common theme to the above questions can be synthesised into the following three key questions which will form the basis of our investigation.

Firstly, how are common DSMs designed and implemented? We wish, in answering this question, to learn more about the kinds of metrics commonly used in text processing, and the sort of difficulties arising when considering how to use them in practice. Secondly, how can we analyse DSMs? In answering this question—which will form the bulk of our project—we wish to discover how DSMs can be compared and ranked relative to different types of document similarity, thus giving us some insight into their performance for a variety of text processing tasks. Thirdly and finally, how can the results of such analysis be leveraged to improve existing DSMs or produce better DSMs?

In order to answer these questions, we designed and implemented a tripartite DSM evaluation system structured as follows: we first constructed a system capable of generating and annotating a structured corpus composed of pairs of documents categorised by the type of similarity associating them, in addition to a gold-standard similarity score; we then built a framework which uses the aforementioned corpus as a testbed for evaluating a variety of DSMs by checking the metric score for each document pair in the corpus against the gold-standard similarity score; and finally we wrote an analysis framework which breaks down the results of the evaluation and produces finer analysis of the results, allowing for a better understanding of different DSM's performances for different similarity types, as well as clearer inter-metric comparison.

The subsidiary goals of this project were therefore to produce an extensible framework for evaluating, comparing and analysing metrics, in addition to our main goal of producing results which might allow us to answer the three main questions discussed above. To present how we have attempted to reach these goals, we have structured this dissertation as follows: in Chapter 2 we will briefly present some of the linguistic and philosophical background to the definition of similarity in order to clarify the concepts we will be dealing with throughout this work; in Chapter 3, we will present the concept of the tripartite framework we constructed in more detail and discuss some issues we considered before beginning the design process; in Chapter 4 we will present the structure of the corpus generation framework, and discuss specific issues with designing the corpus categories we used in our final experimental analysis; in Chapter 5 we will discuss the general structure of DSM evaluation framework, as well as discuss the difficulties faced while implementing the specific metrics used in our experiment (particularly the semantic similarity

metrics, discussed in §§5.4–5.5); and in Chapter 6 we will discuss the design of our evaluation framework, but also present and comment upon the results of our experiment, wherein we aim to discover how well individual DSMs perform for the corpus section corresponding to the sort of similarity they were designed to measure, and whether there are any 'surprises'—*i.e.* DSMs which perform well for similarity types they were not originally designed to deal with. Finally, in the conclusion—Chapter 7—we will review the results of previous chapters, see how far we have come towards reaching the goals and answering the questions set out in the present chapter, and discuss how the results of analysis could be used to improve existing metrics and/or design new metrics capable of dealing with complex kinds of similarity (typically sorts of semantic similarity) in future work.

Additionally, although we will discuss some potential further work in our conclusion, because our project revolved around constructing an *extensible* framework for the construction of a testbed corpus and the evaluation and analysis of DSMs, we will mostly be synthesising the suggestions for future work and further improvements which we will be making throughout this dissertation. The general purpose of this project is thus not just to learn about DSMs and how to improve them, but to condone (and hopefully exemplify) a more rigorous, scientific approach to text processing and computational linguistics as a whole.

Chapter 2

Background and Theory

In this chapter, we wish to clarify the core concepts we will be discussing throughout the dissertation (*cf.* §2.1), in particular that of what 'similarity' could be taken to mean in the context of 'document similarity'. Using such definitions as a basis, we will attempt to clarify what different kinds of document similarity there might be (*cf.* §2.2). Discussion of the theoretical background for individual DSMs, which one might expect to find in this chapter, will instead be provided when discussing individual metrics in Chapter 5. Finally we will attempt to draw conclusions relating the results of the discussion in this chapter to the structure and goals of this project in §2.3.

2.1 Philosophical Foundations

The definition of 'similarity' in the philosophical domain seems as nebulous as the domain itself. Indeed, the absence of a definition for the term in the otherwise-well-rounded *Oxford Dictionary of Philosophy* (Blackburn 1996) may serve as an indication that the term lacks a definite description. However, a broad notion of similarity exists in the philosophical literature since Hellenic times, usually implicitly understood as the fidelity of property-conservation between an object and its reference. However, this broad definition does not always fully fit the notion of similarity involved in DSMs, as we shall see below.

2.1.1 Platonist Similarity

For instance, in Plato's *Republic* (*cf.* Griffith and Ferrari (2000), books VI–VII), it is discussed how what defines a class of objects is the sharing of features of ideal objects. For instance, a chair is similar to another chair *because* they both have properties relating them to an ideal chair which Plato calls the (conceptual) *form* of a chair (*e.g.* back and seat, one or more legs elevating it from ground level, etc.), even though they may differ with regard to other properties (*e.g.* colour, construction material, presence of arm-rests, etc.).

Judging the similarity of entities by examining their sharing of an ideal form may seem intuitive enough, but there are several problems. First, it seems we must commit ourselves to the possibility of abstract/general enough forms for non-trivial comparisons to be made: to illustrate, a stool and a chair are intuitively similar, but if we do not commit to their being a hierarchy of forms in which some ideal amalgamate of the forms of chairs and stools exists, we cannot evaluate their similarity under a such platonist-inspired system.

However, if we *do* allow for such a hierarchy, we must not only specify how *degree of similarity* is to be judged, but more importantly we must also describe limiting conditions for the stipulation of such amalgamations of forms, lest we allow for any two objects to be qualified as similar through the postulation that there exists some ideal amalgamation of their corresponding forms (for example we might say that a chair and a whale are similar because both have essential properties of the ideal form of things that exist physically). This objection, it turns out, is fairly similar one presented by Davidson, which we will discuss below.

2.1.2 Similarity and the Foundation of Mathematics

Later metaphysicalists followed more refined variations on the standard platonist attitude towards similarity, in particular in the area of philosophy of mathematics. Early developments of set-theory, in the work of (Cantor 1874) who describes sets 'a collection of objects taken as a whole', implicitly allow for the grouping of objects according to some identifying properties. This is made more explicit in the development of naïve set-theory underlying Frege's logicism (*cf.* Frege's 1884 *Foundations of Arithmetic* and 1893 *Basic Laws*

of Arithmetic), where sets are carved from a domain of objects according to the truth of second-order logical statements. Since Frege takes the extension of second-order predicates to be concepts (*i.e.* properties), we get the idea that both in language and in mathematics, we can group together entities according to shared properties. This is a more rigorous formulation of what was effectively a platonist idea to begin with.

The core objection to Fregean logicism was, of course, the inconsistency posed for naïve set-theory by Russell's paradox, but revised mathematical set-theories (*e.g.* ZFC) retain the interpretation of set definitions of the format $\{x|logical_conditions(x)\}$[1] as being groupings of (implicitly similar) objects according to shared properties. While this definition evades the rigidity (and metaphysical baggage) of purely platonist definitions, we still are left without a way to cash out a plausible notion of *degree of similarity* (after all, a chair is more similar to a another chair than to a stool, but is more similar to a stool than to a whale), and the problem remains that anything is plausibly similar under *some* criterion of association. These two problems will obviously need to be addressed to reconcile a technical definition of similarity with our linguistic intuitions and practises.

2.1.3 Davidson on Similarity in Metaphors

(Davidson 1978) attempts to respond to one of these qualms by considering the sort of similarity at play in metaphors. This kind of similarity suits our task of clarifying the concept of similarity at play in DSMs well since it is a purely abstract notion of similarity (rather than a specific one, such as the length of words, the meaning of two segments of text, etc.), since metaphors can be built on the basis of any imaginable similarity between the literary construct and its real-world counterpart.

Davidson (*ibid.* pp33–34) frames this sort of abstract linguistic notion of similarity as a pragmatic issue: similarity is the sharing of one or more properties by entities; if we limit our definition to this, then I, for instance, am similar to Tolstoy by virtue of us both having been infants at some point in our lives; therefore any non-"garden variety" notion of similarity must

[1]Namely the set of all elements of a given domain satisfying a set of logical conditions (*e.g.* "x is prime", "x is Greek and Mortal").

be not only the sharing of properties between entities, but specifically the sharing of properties which are relevant to the context of evaluation and use of similarity. Simply put, we select the properties we use to evaluate similarity based on pragmatic goals of similarity evaluation (*e.g.* if we seek to group documents by length, we will look at syntactic features like word-count or character-count, and ignore the particular words used, etc.). However, we still need a conceptual way of defining degree of similarity.

2.1.4 Similarity in the Cognitive Sciences

Relatively more recent work on the pragmatics of similarity assessment have sought to tackle the issue of degree of similarity. In the 1980s cognitive scientists such as Douglas Medin aimed to characterise the relation between properties and degree of similarity (neatly summarised in Gärdenfors, P. (2004, p111)) as follows:

> [...] two objects falling under different concepts are similar because they have many *properties* in common. [...] Medin formulates four criteria for this view on similarity: (1) similarity between two objects increases as a function of the number of properties they share; (2) properties can be treated as independent and additive; (3) the properties determining similarity are all roughly the same level of abstractness; and (4) these similarities are sufficient to describe conceptual structure: a concept is equivalent to its list of properties.

This almost seems to provide us with what we want, in that we now can formulate a more rigorous definition of what similarity within a pragmatic context is: it is the sharing of contextually-relevant properties or *features*, and the degree of similarity between two objects is evaluated based on the degree of correspondence between contextually-significant properties of the object. However, we must also add that some properties may be more important than others in assessing similarity, a point which Medin ignores and perhaps even contradicts with point (3). This point will be important when considering how to improve metrics as will be discussed in the conclusion.

2.1.5 Wittgenstein on Similarity

While the view of similarity we have arrived at in §2.1.4 seems to fit our intuitions about how the term similarity is used in DSMs, there are two other views on how we could consider similarity in DSMs, one of which emerges from Wittgenstein's discussion of 'family resemblance', and the other which is derived from Wittgenstein's language game account of natural language semantics. Both these views are drawn from (Wittgenstein 1953), and may prove to be of interest in our discussion of different kinds of document similarity in §2.2.

(Wittgenstein 1953) considers language to be a form of life (*ibid.* §23), it is a series of "language games" which we develop, acquire and reject as we practise them or enter communities where new language games supplant our own. It is something the terms and structure of which are "not something fixed, given once and for all" (*ibid.*). The key idea is that the meaning of expressions of our language is entirely set by our use of them. An example Wittgenstein (*ibid.* §2) gives is a "complete primitive language"—the *only* language a tribe of builders possesses—entirely composed of the expressions "block!", "pillar!", "slab!" and "beam!". Let us call this language L_T. In L_T, the utterance of such expressions by one individual A to another, B, causes the B to bring a particular type of object (*i.e.* a block, pillar, slab or beam, depending upon the utterance used) to A. We speakers of English can *interpret* "slab!" as meaning "Bring me a slab!". Is this to say that this is what "slab!" means? Recall that L_T is primitive and complete, so that to the tribe speaking only L_T, "slab!" cannot be analysed in such a way, yet they communicate successfully. Therefore we must not give in to the temptation to analyse another language in terms of our own, when the practitioners of that language have ability to apply and understand their language while not possessing ours (*ibid.* §20). What then is the meaning of "slab!"? Wittgenstein posits that in L_T, the fact of "slab!" having the same meaning across the tribe is equivalent to its *having the same use* across the tribe, and nothing else. Wittgenstein (*ibid.* §7) calls all this, "[the] language [L_T] and the actions into which it is woven" a 'language game'.

How is this notion of language game significant for the evaluation of similarity? According to this view, the semantics of a language are graspable through understanding how it used, and hence nothing is hidden, everything

is plain view. This first of all reinforces the pragmatic nature of similarity evaluation as underlined in our discussion of Davidson in §2.1.3, but more importantly, presents similarity between two linguistic entities as being not the sharing of hidden properties, but similarity of use. This attitude towards language will come up again when we discuss semantic vectors in §5.5. This view effectively conflates pragmatics and semantics of language, and additionally makes it more difficult to distinguish semantic properties of text from syntactico-lexical ones (since these also are features of how text is used), an aspect we will discuss further in §2.2.

The second view, that of 'family resemblance', actually stems from Wittgenstein's clarification of what a language game is (*cf.* Wittgenstein (1953, §§65–75)). Wittgenstein (*ibid.* §65) claims that there is no such thing as the 'general form of propositions of language', and that language is in fact a set of acts *related* to each other in a variety of ways. In doing so, he is calling upon an abstract notion of similarity, which he clarifies in the statements that follow—principally §66. Here he describes family resemblance as "a complicated network of similarities overlapping and criss-crossing: sometimes overall similarities, sometimes similarities in detail". For instance it is hard to say what the essence of a game is, namely how all games are similar. After all, poker is not like football, but is like blackjack; some games have rules, others (such as those played by children) operate without the need for determinate guidelines; some games have goals and finishing conditions, others (*e.g.* juggling a hackey-sack) have no 'purpose' and can be played forever. To search for a common property to fit our intuition that these all share some fundamental property of being a game would tell us little about how we come to understand that property, or even if it exists. In fact, it seems more intuitive to claim that it is the sharing of the same kind of similarity with several other games that makes each game part of this network, even though it may not share that similarity with all games on the network.

We retain from this that complex forms of document similarity may be—if we hold Wittgenstein's position—quite different from that we had conceived of in §§2.1.3–2.1.4, above. Instead of two objects being similar by virtue shared properties or features, they could be considered similar by virtue of holding the same broad kind of similarity to some intermediate objects, which recursively may only be similar to one another by the same transitive notion of similarity. This appears to be a more complicated view than the first Wittgensteinian view described earlier to conceive of in practice, but is will

be relevant to our discussion in §2.3.

2.2 Different kinds of Document Similarity

In §§2.1.3–2.1.4, we arrived at a plausible definition of the sort of similarity we believe to be at play in DSMs, namely that sharing features relevant to the nature of comparison is what determines the similarity between documents, and the frequency of observation of such shared features is used to determine degree of similarity. This definition is that of an abstract, general notion of similarity, that we will now attempt to add granularity to this overall picture of similarity by considering its relation to more specific kinds of similarity used in text processing, and what problems may arise during such consideration.

2.2.1 Three Classes of Document Similarity

In the linguistics (both computational and non-computational) literature, it is common to classify features of text documents into three broad feature classes, namely syntactic, semantic, and lexical.[2] Naturally, segments of language can have other feature classes depending on the media, for example spoken words have phonetic features on top of the above three, but we shall restrict our considerations to textual features.

Syntactic aspects of language cover how we form correct sentences and the superficial rules that govern their organisation, namely relating to grammar and the structural relations between words (for example how an adjective usually qualifies the noun that follows it most immediately). Lexical aspects of language govern the grouping of words by theme, usually also involving some notion of hypernym/hyponym-defined hierarchy of terms[3]. Finally semantic

[2] Certain authors such as (Allen 1987, pp6–8) also add to this set of feature classes the class of pragmatic features, under which we consider how words and expressions contained in a document are used. These are, without a doubt, important features when determining the meaning of text, but for that reason precisely I would consider such features to be semantic.

[3] 'Feline' is a hypernym of both 'lynx' and 'cat' (and others), which are its hyponyms, because it holds an ISA (literally 'is a') relation to them in that a cat *is a* feline, equally

aspects of language are the most complex, and qualify how we understand the meaning of our utterances and writings. We shall have more opportunity to understand how these broad classes of similarity are illustrated when we examine their features, below.

According to the Davidsonian view, and considering the three classes of document features presented above, we might wish to represent the organisation of the various types of document similarity and their relation to the classes and properties as shown in Figure 2.1. We now discuss how features/properties

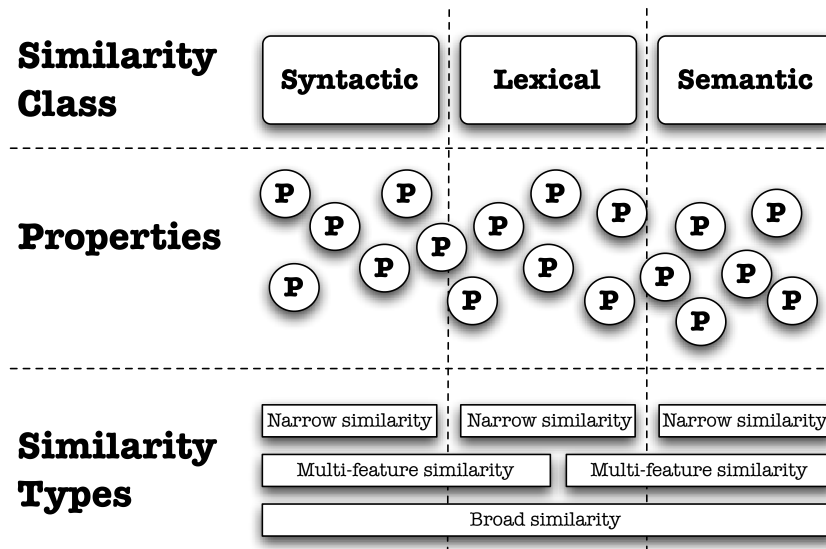


Figure 2.1: Classes, Properties and Types of Document Similarity

of documents could be clustered and fall into the broad classes of similarity we have just defined.

2.2.2 Similarity Features

Consider features such as word-length, the presence or absence of individual parts of speech or of particular words, all of these could be considered to be clearly syntactic properties. To exemplify lexical features, we would consider

a lynx *is a* feline, but it makes little sense to say that a feline is a cat or a lynx.

document tokens belonging to particular lexical groups (*e.g.* a document containing words such as 'drive', 'wheel', 'motor' would have the property of having member-words of the 'mechanical' or 'automobile' lexical groups, amongst others).

As for what constitutes semantic properties, specifying what exactly qualifies is undoubtedly a philosophical matter; however we can suggest features such as the truth value of the propositions within, implicated propositions[4], sense and reference of words in the document[5], and other such features related to the meaning of words and expressions constituting the documents being compared.

The difficulty in coming up with specific examples for clearly semantic features of language serves as an early indication that determining whether features fall clearly in a category is not always obvious. We must therefore also consider the case of cross-category features—features that do not cleanly fall into one category or another.

2.2.3 Cross-Category Similarity Features

Certain tokens representing document properties in Figure 2.1 have been shown overlapping the boundaries between similarity classes. This is because some document features one can conceive of fall into categories ambiguously. For instance, we consider the presence or absence or particular terms in a document to be a syntactic feature, yet the presence of groups of terms holding some lexical relation to be a lexical feature. Thus evidently there is

[4] The implicated propositions are taken to be the contextually-determined subtext of component sentences of a document, as discussed by (Grice 1975), and makes use of what (Allen 1987, p6) refers to as "pragmatic knowledge". For example, if when asked "May I borrow your stapler?" one replies "My office door is unlocked", one is in fact also stating something akin to a positive response to the question, despite the actual reply having no obvious thematic relation to staplers or permission.

[5] We here take the notion of sense/reference distinction to be that first presented by (Frege 1948)—originally in 1892—where the reference of a term or expression is the real-world object or concept it 'point to', and the sense is its mode of presentation, *i.e.* the way by which we come to grasp the reference. For example both 'Superman' and 'Clark Kent' indicate the same individual, and therefore share reference. However they are have a different sense, exemplified by the fact I can believe 'Superman can fly' without believing 'Clark Kent can fly'.

a syntactic aspect to some lexical features, and possibly vice-versa.

Likewise, lexical relations within a text play some role in determining the meaning of the text, thus semantic features of texts arguably rest upon some lexical features, and vice-versa since lexical groupings are intuitively understandable in the terms of the relation between the meaning of the terms: a regular speaker of English can associate 'car' with 'engine', 'steering wheel' and 'drive' rather than with 'number' or 'solar system' because he understands the meanings of the words and knows them to bear some closer ontological relation than words external to the lexical group of automobile parts.

This last point will be important when we consider the effectiveness of Wordnet-based metrics in §5.4 and of distributional approaches to semantic similarity in §5.5, following the dictum of (Firth 1957) that "you shall know a word by the company it keeps", a sentiment which appears fairly similar to the above point about the relation between lexical features and semantic features. However, before that we must discuss how such cross-category features feed into the larger problem of cross-category similarity types.

2.2.4 Classifying Similarity Types

Since, as discussed throughout §2.1, we have taken the evaluation of document similarity to be based on identifying and exploiting the identifying features of the kind of similarity we wish to measure, it is clear that the different specific kinds of document similarity we will encounter during real-word text processing tasks will be built upon the properties discussed above, and thus will also share the same problems when it comes to classifying them into the three broad similarity classes presented in Figure 2.1 and §2.2.1.

Naturally, there are specific notions of document similarity that fit snugly within the confines of the three similarity classes. There is no need for—or possibility of—appealing to semantic or lexical properties when attempting to compare the word length or syntactic differences between two texts.

Likewise, raw lexical comparison is devoid of any need for semantic comprehension or syntactic analysis, since for example the strings "The big ball is red, not blue." and "Not the big blue ball, the red ball!" are fairly different syntactically (this is not to say there is no syntactic relation between them,

just that it is not what we call upon to compare them in this context) yet are lexically similar according to any lexical metric treating strings which can be compiled into identical lexicons as being lexically identical.

Finally, we can conceive of examples where semantic similarity can be evaluated without direct appeal to syntactic or lexical relations between the compared documents. For example, the English sentence "the colour of crystallised frozen water falling from the sky is a saturation of primary colours" and the Japanese sentence transliterated as "`yuki-wa shiroi desu`" ("snow is white") share the same proposition while being lexically and syntactically disassociated (since both the lexemes and the sentence syntax differ between languages and sentences). A more radical example yet is the semantic correspondence between a sentence and its logical form, such as the relation between "if Socrates is Greek the Socrates is mortal" and "$Gs \supset Ms$" (for the valuation $G =$'is Greek', $M =$'is mortal', $s =$'Socrates' and $\supset$ as logical consequence).

The above examples are not idealised cases, since each corresponds to a kind of document similarity called upon in real-world text-processing tasks (for instance: measuring the size of a document, evaluating lexical correspondence, and translating from Japanese to English, respectively). Nonetheless they are not representative of the usual admixture of similarity types one would expect to find in specific notions of document similarity associated with a variety of text-processing tasks.

Syntactic and lexical features of two documents may be exploited to determine topical relation between them, as exemplified by distributional models of semantic similarity inspired by the ubiquitous quote by (Firth 1957) first mentioned in §2.2.3. Effectively, we evaluate the lexical similarity of the linguistic context of two words to attempt to determine their semantic similarity, possibly determining context according to syntactic features such as grammatical relations, as suggested by (Grefenstette 1992), for instance.

Similarly, while part-of-speech (POS) tagging can be performed using purely stochastic methods, linguistics students may also be familiar with the practice of considering what other words may take the place of a word they are attempting to tag in order to aid manual POS-tagging. Thus the task of comparing the syntax of two sentences could, conceivably, be lexically motivated, rendering such an obviously syntactic notion of document similarity (at least) partially lexical as well.

Finally, while lexical comparison of sentences of different languages may be treated automatically using translation dictionaries or aligned parallel corpora, human translation frequently calls upon semantic knowledge of the source and target languages, especially when a word in the source sentence is unusual, ambiguous or unknown to the translator: we can nonetheless use natural language understanding of the rest of the sentence to derive the use of the word and attempt to evaluate lexical relations as a result.

The point being made here is that many text processing tasks involve a specific notion of document similarity which does not neatly fall into the three broad classes of document similarity we defined above. If we are to—as the philosophical background discussed earlier suggests—treat the work of DSMs as identifying contextually-relevant key features of documents and exploiting them to determine similarity between them, then how can we account for the difficulty in classifying certain (typically more complex) kinds of similarity? The answer we propose in §2.3 is that such a classification problem is not so much of a problem, but rather a feature when it comes to the goal of this project.

2.3 Dropping the Class Distinction

In §2.1.5 we discussed a Wittgensteinian alternative to the position we derived from Davidson's discussion of similarity. The two—possibly compatible—views we presented were the following: the first stated that the semantics of natural language were entirely determined by the uses of our expressions—ergo that in short, one could not purely distinguish semantic properties from from syntactico-lexical properties, since semantic features reside one the 'surface' of language rather than being hidden from view; the second was that similarity between two entities did not always have to be the direct sharing of properties, but is sometimes determined by both entities being similar to some same body of entities, being part of a the same network of similarity, without being evidently similar to each other (as could be determined by the sharing of properties).

Our problems in §2.2 with determining the classification of more complex types of similarity, and even of more complex properties (namely those relating the meaning, but also possessing syntactic and lexical aspects) seem

to give some weight to these Wittgensteinian views being a picture of document similarity more in line with our intuitions. If this is true, what are the consequences for our project, our goal of better understanding how metrics work in order to improve them?

The claim we make here to conclude this chapter is that considering this new view will guide us in the following three ways. First, it will help us understand why Wordnet-based (*cf.* §5.4) and vector-based (*cf.* §5.5) metrics can be successful in computing semantic similarity, as hinted at at the end of §2.1.5.

Second, it gives better justification for our project of measuring the success of a variety of metrics against a testbed containing a variety of similarity types. If we were to observe the abnormal success of a metric designed for syntactic comparison, rather than write off such results as flukes or abnormalities, we instead would find justification in that the document features it exploits to determine similarity are not only syntactic (as they might appear at first blush), but also plausibly semantic features, since the Wittgensteinian view discourages the strict classification of such features.

Third and finally, since according to Wittgenstein some complex forms of similarity may be composite, constructed from the overlap or networking of other forms of similarity, then we have some justification for one of the final goals of this project: to provide methods for creating better metrics through the construction of hybrid metrics, constructed according to the results of our analysis.

In the following chapters, we will present the project we designed after considering the Wittgensteinian view, alongside results we believe will confirm the theoretical discussion presented in this chapter.

Chapter 3

Methodology

In the introduction, we first presented the goal of the project: to evaluate a variety of DSMs using a testbed containing documents paired according to a variety of similarity types, and to analyse and compare them based on the results of the evaluation. This, we stated, required the construction of a framework—which we wanted to be as extensible as possible in order for it to useful for future work—which would generate the testbed corpus, run the metric evaluations, and analyse the results. In this chapter, we will discuss the general design goals we considered before constructing the framework, and problems we had to take into account. A more detailed account of the implementation of such a framework and implementation problems will be presented in Chapters 4–6.

The first step towards analysis we shall discuss (§3.1) is that of constructing a corpus. We will present the similarity types such a corpus would ideally have for our experiment, the structure of such a corpus as well as methods used to increase the volume of the corpus without the need for new sources. The second step (§3.2) we shall discuss is the task of evaluating DSMs, alongside the general problems one would expect to encounter during implementation, the format of the evaluation output, and scientific aspects such as result normalisation. The third and final step (§3.3) is the analysis itself. We shall discuss what methods were considered for interpreting evaluation results, and what methods were used to get finer granularity in comparing and evaluating metrics.

3.1 Constructing a Corpus

Constructing a testbed corpus was a crucial part of the experiment. The richness of the results depended upon the variety of similarity types we could get involved. Additionally, we would have to consider how to format the corpus so that as much information as possible could be processed in advance, and that accessing the information for different kinds of similarity would 'look the same' so that metric evaluation scripts would not need to be tailor-made for individual sections of the corpus. In this section, we will discuss what factors would affect the design of the corpus generation framework we will present in detail in Chapter 4, and what requirements we needed to keep in mind while designing it.

3.1.1 Selecting Document Similarity Types

Obviously, there is an extremely wide range of subtly different notions of document similarity to be concerned with. Not only does attempting to test a selection of DSMs against *every* or even *most* seem intractable, it is also a simply unrealistic task since by the Davidsonian notion of similarity discussed in §2.1, if any identifying property can be used to draw similarity between two entities, then we arrive at a trivial notion of similarity. However even when we restrict the notion of similarity at play in document similarity to that built upon document properties related to the pragmatic context of comparison, it would be difficult to build a definitive list of document similarity types against which to evaluate metrics, seeing how there is no definitive list of pragmatic contexts in which document similarity may be used (indeed, new applications of DSMs come up with new applications in text and language processing).

To render the experiment interesting, the goal should therefore be to collect document similarity types associated with popular DSMs, rather than aim for the construction of an exhaustive list—a task which may not even be possible to complete. In this section, we will list the types of specific document similarity used in this experiment, discuss a few of the applications in which they play a role, before discussing how we implement them in the evaluation corpus in Chapter 4.

Despite the fact that we presented and argued for, in §2.1.5 and §2.3 a Wittgensteinian account of similarity which denies the existence of any deep divides between semantic, syntactic, and lexical kinds of similarity, we do not deny that similarity types can appear to fall into one class of similarity more than another on a superficial level; and equally DSMs are designed to deal with similarity of a certain class, although the whole point of §2.3 was to show that there was theoretical justification for DSMs performing better than expected in evaluating other similarity types from other classes. As a result, we wanted to have, in our experiment, a few similarity types from all three main classes of similarity (syntactic, lexical and semantic), if only for the purpose of having a corpus such that for each metric, there is at least one section where it is expected to perform decently well (since that section would associate documents based on the similarity type the metric was designed for, or something close).

Therefore we wished to have one or more corpus sections where documents were paired by syntactic relationships, based on superficial differences such as random edits, or on different words but with similar grammatical structure. We built corpus generating classed for both these kinds of similarities, although the resulting document pairs are not always realistic examples of English due to the crude methods of generation. We will discuss such syntactic 'toy' corpora in §4.2.

We also needed some lexical corpus sections. As we will discuss below, most of the semantic sections of the testbed corpus also contain lexical relations between the document pairs (sometimes as a factor of similarity, sometimes not). However, we aimed to have at least one section where documents were associated on 'purely' lexical grounds without obvious semantic factors. We obtained this by grouping sets of paragraphs from different literary genres. We shall discuss the implementation details in §4.3.

Finally, we wanted a set of corpus sections to represent a variety of semantic similarity types. Here is an account of three such types used in our experiment.

Paraphrase is a form of semantic similarity, since it exhibits interesting syntactic features (length and grammatical structure) as well as lexical attributes (similar choice of words). We sourced three different forms of paraphrase

(intentionally-generated paraphrase[1], and two different types of translation-generated paraphrase[2]) to create a large corpus section which could be split into three sub-sections, using sources provided by another project (*cf.* Cohn et al. (2008), Dolan et al. (2004)). We shall discuss the implementation of this part of the corpus in §4.5.

We wanted to include another type of semantic similarity which also exhibited lexical/semantic relations between the documents, as well as a kind of syntactic relation different from that present in paraphrase. This kind of semantic similarity is that present in summarisation, where one document is shorter than the other, but presents the core meaning of the larger document. For this section, we match academic articles with their abstracts, using only articles from the same subject-area so as to minimise the role of lexical relations in the corpus (since papers from the same subject area all share a lexical relation, the rendering lexical features less important in the evaluation of similarity). We will describe this part of the corpus in §4.6.

The last kind of semantic similarity we wished to include in our corpus is a bit harder to describe. We wanted to choose documents about the exact same topic, and which therefore could be paired up according to lexical and semantic features while being structured in a different manner. If you will, we wanted a counter-part to the abstract-article pair section. We consider using academic papers paired up by *specific* topic (*e.g.* two papers on Lewis' modal realism, two papers on the inconsistency of Robinson arithmetic, etc.), but it quickly became clear that doing so would involve a lot of manual processing, since most pairs would have to be checked by a human annotator. The solution we came up with was to use for our document pairs the English version of a Wikipedia article paired with the Simple English[3] version. This was an ideal source since the documents were pre-matched (*i.e.* we could find the Simple English version of an article, if it existed, from the regular English version of the article), and the document pairs bear a loose semantico-lexical

[1] By 'intentionally-generated paraphrase' we mean a pair of documents where one was directly written to be a paraphrase of the other.

[2] By 'translation-generated paraphrase' we mean document pairs where both documents are English translations of a foreign phrase, each document therefore being an 'indirect' paraphrase of the other.

[3] Simple English Wikipedia (`http://simple.wikipedia.org/`) is a version of Wikipedia where many English articles are reproduced using shorter sentences and simpler vocabulary, aimed at children and individuals in the early stages of learning English.

relation to each other, whereas syntactic aspects are quasi-irrelevant since the articles can be written quite differently, and the difference in length varies. We describe the implementation of this part of the corpus in §4.4.

3.1.2 Designing a Corpus Construction Framework

As stated in the chapter introduction, the twin goals of our project were both to construct a framework capable of generating the particular corpus for running an experiment from which we could derive some novel conclusion, and to do so while keeping in mind that the work—if successful—could be reproduced for different (or a larger set of) metrics, and for larger, more complex corpora. In short, our corpus generation system should be modular and easily extensible. We will discuss how we achieved implementation of a general extensible framework in §4.1.

But before any implementation work was to be done, we had to think about the format and structure of such a corpus. Since we want to create a testbed against which to evaluate DSMs, with the goal of telling us how well each metric works for each kind of document similarity featured in the corpus, we naturally have to divide the corpus into sections, where each section contains documents paired according to a certain type of document similarity. We then had to think about what corpus sections would look like. The idea was not only to associate documents according to a type of similarity, but also to give some indication of *how similar they were* relative to the similarity type characterising the section the pair was in.

Each entry in the sections of the testbed corpus would therefore be a triplet, the first two members of which are the pair of documents the similarity of which is being determined, and the third member is a gold-standard similarity score which is an 'absolute' degree of similarity between the documents, and should be a value between 0.0 and 1.0. We will explain the notion of gold-standard in a bit more detail in §3.1.3 below. The gold-standard is therefore important, as two documents may be paired in different sections of the corpus, but must be considered different entries from one another since the gold-standard score is set relative to the type of similarity defining each section of the testbed.

The possibility that the gold-standard score for a document might be 0.0

(complete dissimilarity) also means we have the ability to match up documents that are dissimilar in any category. Indeed, we *must* do this, since if all documents are matched (high similarity score) in some section, a dummy metric always stating that documents are similar will nearly always be correct (*i.e.* in agreement with the gold-standard score). We therefore need a mechanism by which to balance each section of the corpus in terms of high-score/low-score document pairs. I'll present a general mechanism by which to do this in §3.1.4.

3.1.3 Setting the Gold-Standard

One of the essential elements in each corpus entry, in addition to the two documents paired up according to some notion of similarity, is the gold-standard score, which we take to be an objective representation of the similarity between the two documents in the entry, against which we shall evaluate the metrics (as will be explained in §3.2). As stated above, we conceive of it as a real value between 0.0 and 1.0, allowing for pairing of dissimilar or vaguely similar documents.

An example entry in a hypothetical "word-count" section of the testbed—where similarity between documents is determined by the difference in word-count between the documents (lower difference, higher similarity)—might be represented[4] as follows (using pseudo-XML tags):

```
<doc1 ID=id1> The quick brown fox jumped over the lazy
dog .  </doc1>
<doc2 ID=id1> I saw a film that was not very good .  </doc2>
<gold_standard ID=id1> 1.0 </gold_standard>
```

Here the gold-standard is 1.0 since both documents of the entry (identified by a unique identifier `id1`) have the same word-count. In contrast, some other entry in the same section might look like:

```
<doc1 ID=id24> The quick brown fox jumped over the lazy
dog in the middle of the breezy afternoon .  </doc1>
```

[4] In practice, we only represent the entries like this in order to share the corpus. For the duration of the experiment, the data will be contained in Python objects, and we will not interact with the entries directly.

```
<doc2 ID=id24> I am here .  </doc2>
<gold_standard ID=id24> 0.325 </gold_standard>
```

In this case, the objective similarity is given to be much lower since there is a great disparity between the word-count of the first document and that of the second.

One might ask how we obtain the gold-standard score. The methods for setting the objective similarity score are essentially up to the experimenter, but in most cases, some heuristic will be used (as is the case in this experiment). For instance, when we create a section matching academic articles with their abstracts as an instance of summarisation, we can simply assume that each abstract is the best summarisation (amongst all the abstracts we have) of the article it was provided with in the first place, and thus we assume a similarity of 1.0 for each original abstract-article pair. To obtain mismatched (low objective score) pairs we will use another heuristic described in §3.1.4. In some other corpus sections, we used a more sophisticated heuristic to obtain a more subtle gradation of objective scores. We will discuss such heuristics in Chapter 4. Ultimately, the most precise way to obtain an objective score would be to have trained scorers familiar with what the kind of similarity at play is go through the corpus and set the objective scores manually. However, not only is this tedious, it may also not always be correct, since as discussed in Chapter 2, some notions of similarity are difficult to define and classify precisely.

The use of heuristics to set the gold-standard scores—acting as objective reference similarity measures for the evaluation phase—may also cause some worry. This is a reasonable objection, but we will discuss the merits of this approach relative to other options in §3.3.1, once the evaluation and analysis procedures are clearer.

3.1.4 Mixing Things Up (If Needed)

As mentioned earlier, if we are using a heuristic to determine the gold-standard score for document pairs, especially if we are using a simple one which exploits the structure of the source data from which we construct corpus sections, assuming matched documents from source data to have a gold-standard score of 1.0 (*cf.* abstract-article pair example in §3.1.3), then

we are in danger of having a skewed score distribution where a majority (if not all) document pairs in certain sections of the corpus have a high objective similarity scores. Imagine now a metric which latches onto some feature which all documents of such an unbalanced corpus section possess, and as a result, giving each document pair a high metric similarity score; the result would be that the metric score would be close to the objective similarity score almost every time (which, as we will discuss in §3.3, would give the metric an unwarrantedly high rank).

To prevent such a thing from happening, and simultaneously to increase the size of our testbed corpus, we can apply another heuristic to add a bit of diversity to skewed corpus sections. The heuristic is similar to the one we used, when assuming that document pairs that are matched in the source material (*e.g.* the abstract for an article) have highest objective similarity (*i.e.* 1.0), we now assume that by randomly mixing elements of matching pairs to form non-matching pairs that the similarity of the thus-obtained mismatched pairs must be minimal.

For instance, in the abstract-article section of the corpus, we have n entries of abstract-article pairs given a gold-standard score of 1.0. Now we can easily generate an additional n entries of abstract-article pairs with a gold-standard score of 0.0. By taking all the abstract documents we have and all the article documents we have, and randomly assigning an abstract to an article (obviously checking that we don't assign it to the correct article which it originally came from). We simply assume that an abstract assigned to a random article will actually have little relation to that article, and not be a summary of its contents. It *could* happen that two articles in our source corpora are close enough for the abstract to actually vaguely match the article, but the odds of this are low, and if our corpus is large enough, such occurrences should not affect the overall results.

However there are certain cases where we must not use this method. For example, if our method of generating the corpus already creates an even distribution for a section, which we will take the case when:

$$\sum_{i \in \text{section_IDs}} \text{gold_standard}(i) \approx 0.5$$

then obviously generating any number of entries with an arbitrary gold-standard score of 0.0 will skew the distribution of scores towards 0.0. Ad-

ditionally, the odds of randomly pairing up similar documents (especially in syntactic sections) are non-negligible, and in these cases expanding the corpus through random mismatching should be proscribed. We will discuss how to regulate this corpus expansion step when discussing the implementation of the general framework in §4.1.

3.2 Evaluating Metrics: Considerations and Concerns

The evaluation section of the experiment is the second large part of the analysis process. Once more, we are concerned with the creation of an extensible evaluation framework which will, for each entry in the corpus test, evaluate the similarity of the document pair in the entry using the provided metrics, and, for each metric, record the closeness to the gold-standard score (*i.e.* to the objective similarity score)—which we will call the CGS score—of the metric's determined score for that document pair.

As was the case for our testbed corpus, we wished to collect a wide variety of DSMs in order to produce rich and interesting results for our experiment. There are, as one may expect, a large number of exotic DSMs available, so we focussed on metrics which seemed to come up most frequently in the literature, or be discussed in highly-quoted papers.

We first assembled a selection of purely syntactic metrics, of which we will briefly discuss the theory and implementation later, in §5.2. These include word and character-count-based metrics, and well known distance metrics (Jaccard distance, Levenshtein edit distance, etc.).

We implemented the BLEU metric from the world of machine translation, discussed in §5.3, which exploits both syntactic/structural and lexical features.

We assembled a collection of DSMs making use of the Wordnet corpus to determine lexical similarity. These will be discussed in §5.4. It will be interesting to see, later on, how these perform on semantic similarity evaluation tasks, bringing us back to the Wittgensteinian idea that there exist no deep divide between the similarity classes.

Finally we collected a semantic similarity metric based on the distributional approach to semantic evaluation. We will discuss the theory and implementation of this approach in §5.5.

However before anything else, we will, in this section, describe the general problems we had to consider before beginning the implementation phase which we will discuss in Chapter 5.

3.2.1 Dealing with the Corpus

The first thing we need to consider is how DSMs will deal with the corpus passed onto them by the corpus construction framework. This is a fairly trivial matter, since the structure of the corpus is fairly simple: two strings representing the document pair, and a float value representing the gold-standard score. While it may look like nothing needs to be done, consider the following two sentences:

1. “The man, dressed in black, walked onto the stage.”
2. “The man , dressed in black , walked onto the stage .”

The difference may seem stylistic, but the first one has 49 characters, the second has 52; the first sentence has 9 words, the second has 12. These may seem like trivial differences to a human reader, but to a simple word counter the numerical difference, especially across a longer document, is non-negligible. Despite word-count and character-count being fairly superficial aspects of a document, we are testing a wide variety of metrics, and therefore cannot prejudge what matters and what does not. This is especially important since we might end up wanting to consider ‘superficial’ syntactic metrics for the construction of hybrid metrics, as will be discussed in the conclusion, so we must normalise the corpus before presenting it to the metrics. This is a fairly simple task, albeit necessary. We shall explain how we did it and what other non-crucial pre-processing tasks we did before testing the metrics in §5.1.

Having addressed the above problems, we run DSMs against the testbed corpus as discussed in the introduction to this chapter. For each DSM *metric* and for each testbed corpus entry *entry* the *metric closeness to the gold-standard* (CGS score) is evaluated using the following equation:

$$\mathbf{CGS}(metric, entry) = 1.0 - |\mathbf{GSS}(entry) - \mathbf{score}(metric, doc1(entry), doc2(entry))|$$

where $\mathbf{score}(doc1(entry), doc2(entry))$ is the *normalised* result of measuring the similarity of the entry's documents ($doc1(entry)$ and $doc2(entry)$) using the DSM *metric*; $\mathbf{GSS}(entry)$ is the gold-standard score for $doc1(entry)$ and $doc2(entry)$ in this entry of the corpus. Therefore, when the DSM *metric* scores very closely to the gold-standard score, then $|\mathbf{GSS}(entry) - \mathbf{score}(metric, doc1(entry), doc2(entry))|$ tends towards 0.0, and the CGS score of the metric ($\mathbf{CGS}(metric, entry)$) for that corpus entry tends towards 1.0.

The CGS score is what we will use during analysis to rank metrics, as will be discussed in §3.3. However before we move onto this discussion, we take note that the score provided by $\mathbf{score}(metric, doc1(entry), doc2(entry))$ needs to be normalised in order to be compared to the gold-standard score $\mathbf{GSS}(entry)$. We will discuss how we go about this in the following section §3.2.2.

3.2.2 Normalising the Results

For each corpus entry the gold-standard score—symbolising the objective, 'correct' assessment of the similarity between the two documents according to the notion of similarity defining the testbed section in question—is a value between 0.0 and 1.0. In order to derive the CGS score per entry as defined in §3.2.1, we need each DSM to produce a similarity score between 0.0 and 1.0. Some metrics already do this, or score on a scale which is reducible to the same scale as the gold-standard score by multiplying the score by some constant factor.

However other DSMs, especially syntactic metrics, produce simple numerical output, (for example the difference between the word-count of two documents) which can vary greatly. For such DSMs, we had to consider how to normalise the results in a sensible manner, as will be discussed in Chapter 5.

3.2.3 Adapting Word/Character-based Metrics

One more thing we had to consider was how to deal with DSMs which evaluate the similarity between tokens, rather than documents: a fair few of the semantic and lexical similarity metrics we collected compare *words* rather

than sentences. In order to compare these DSMs with other DSMs, capable of computing a similarity score for sentences, paragraphs, or even entire articles, we needed to determine a way to scale these DSMs from word-based to a string-based metrics. In short, we had to write 'wrappers' that would use the word-based DSM and some scoring heuristic to obtain a similarity score for documents of any length.

We had reservations about doing this because it is hard to say whether we are evaluating the original DSM or our adaptation. Since the heterogeneous nature of DSMs is part of the motivation for this entire project in the first place, it became clear we would have to accept this issue as a necessary evil. However, we did want our adaptation to interfere as little as possible with the way the original DSM runs. As such, we usually opted for the simplest scoring heuristic giving good results, rather than attempt to construct sophisticated heuristics which might give better rankings, but places more emphasis on how the heuristic works than how the original DSM performs.

We will discuss the implementation of such heuristics in §5.4 and §5.5.

3.3 Analysis: The Devil in the Details

The final step of the experiment is to get results. To do this, we wrote an analysis script which, in line with the rest of the project, had to be general enough to be usable for further work (*i.e.* different/bigger testbed corpus, different metrics) without any (significant) modifications. As was the case for the corpus construction and the metric evaluation, we will present the implementation details in a later chapter (Chapter 6), alongside the actual results of our experiment. Here we will discuss the general idea behind the design of our evaluation script, and how we planned to rank and compare the metrics.

In §3.2.1 we discussed how to calculate the *metric closeness to the gold-standard* (CGS) for each entry in the corpus. This CGS score, a value between 0.0 and 1.0, corresponds to how similar—for a particular entry—the metric's evaluation of document similarity was to the objective standard set during corpus construction; higher scores means the metric's evaluation is correct for that particular corpus entry, lower scores means it differs greatly

from the gold-standard score.

In running the experiment, we collect the CGS score for all entries of each section of the corpus, for every DSM. Thus, for every metric we can get a metric score for each corpus section by calculating the average CGS for the entries of that section. This allows us to get rankings for each section of the testbed corpus, where metrics with a higher average CGS score for some section are considered to be more effective DSMs for the kind of similarity characterising that section than lower-scoring metrics.

This may seem simple and effective, but two questions come to mind. First, is this the best way to go about evaluating metrics? Second, while this gives us definite rankings, the basis for comparison may seem a little superficial. Is there no way to obtain more detailed information while ranking and comparing metrics? We will attempt to respond to both questions below.

3.3.1 Gold-Standard Ranking vs. Retrieval

Is using average CGS per section the best method for ranking and comparing metrics? It is hard to answer such a question, since one would need to exhaust all other options, including ones we had not considered. The idea behind this approach is fairly intuitive in the first place: we see how well each DSM's evaluation of similarity compares to that of an ideal metric (as represented by the gold-standard score). Naturally the actual gold-standard score is usually imperfect, since we have used heuristics to set it and expect anyone constructing a large testbed corpus would have to do the same, but the heuristics we have used are, we claim, reasonable (as discussed in §3.1.3); therefore the imperfection is intended to be mostly negligible for the purpose of our analysis, since all metrics are tested against the same gold-standard scores.

Another option that came to mind when we were considering how to go about analysing and comparing metrics was to use the following protocol inspired from information retrieval. We would first match up all the similar document pairs in the corpus, which would be divided up into sections as done previously. For the metric evaluation, for each section, we would randomly select a document (*not* a document pair) from the pairs in that section and call it the query, compute the similarity of the query with all other documents

in the pairs of the section and rank them according to metric similarity score for each DSM, and then see where the document originally paired with the query appears in the rankings for each DSM. Using this information, we can calculate the precision, recall, and F-measure for each metric. By repeating this operation a fixed number of times (presumably depending on the desired granularity of results and size of the corpus) for each corpus section, we would obtain average precisions/recalls/F-measures per metric, per corpus section, and use these to rank the metrics for each corpus section (high F-measures would signify better metrics, since they are more successful at reproducing the original pairings).

Why did we not use this method? There are three good reasons. The first reason is that it is computationally more expensive (unless some clever heuristic is used). Where n is the number of times a document was randomly picked for a corpus section, and k is the number of documents in that section, then the number of similarity measurements is obviously $n \cdot k$ per metric. Whereas when using the gold-standard-based evaluation method, we only need k comparisons per section, per metric. Bearing in mind that evaluating some metrics (namely the Wordnet-based DSMs) took over half a day in our experiment, multiplying that by some significant value makes the experiment impractical.

The second reason is that in some sections of the corpus—namely the syntactic sections—some of the documents paired up are very similar. Therefore running syntactic DSMs, which one would expect to perform well in such sections, through this information retrieval-like evaluation system would produce bad results, since many very similar documents might rank highly while not being the documents originally paired with the query. This is less of a problem for semantic/lexical sections and metrics, but since we want to evaluate all metrics we have against all the types of similarity present in the corpus in order to compare DSMs against the testbed corpus' sections, it makes no sense to adopt a method which will produce accurate evaluation results only for some metrics, and some sections of the corpus.

The third and final reason we used the gold-standard method is that it allowed for a more fine-tuned metric analysis, as will be described in §3.3.2, while the information retrieval-based method does not allow this, as should soon become clear.

3.3.2 Finer-grain Analysis: Breaking Down Results

Earlier we asked how we could get more precise results than simply ranking metrics by section using average CGS scores. The solution we propose is as follows, and involves two methods.

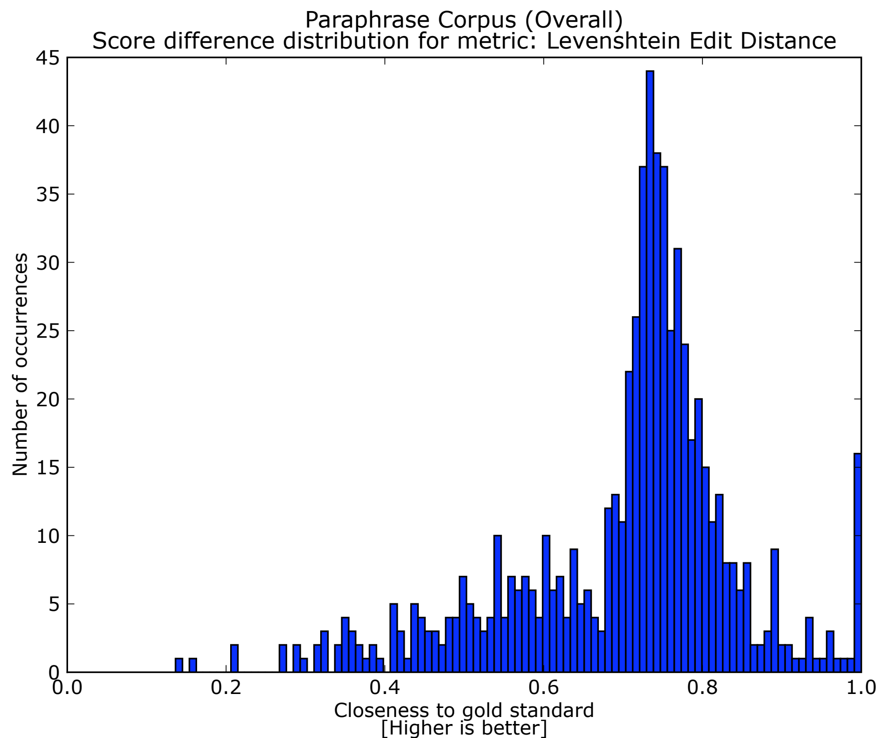


Figure 3.1: Example of Visually Represented CGS Score Distribution

The first method we propose involves visually representing the distribution of CGS scores for each metric, for each section, as shown in Figure 3.1. When the scores are consolidated into sharp spikes, as seen in the figure, then we can easily see that the metric gives consistent scores. When the spikes are towards the right end of the plot, the DSM is generally successful, and when it is towards the left, the DSM is poorly adapted for measuring the kind of similarity defining the corpus section being used for evaluation. Likewise, when we observe disjoint spikes with some distance between them, then we can state that the set of features the metric is using for evaluation is present

throughout the corpus section, but is most likely not used in identifying the kind of similarity defining that section of the corpus, since the features are identified in similar pairs as well as dissimilar pairs. If there is a complete absence of spike or concentration of results, then the metric is evaluating similarity quasi-randomly[5].

Of course, there will be cases where a DSM will have a mix of the above features. For example, for some section of the corpus, metric A might rank higher than metric B in terms of its average CGS score; however it might be, upon looking at the distribution of CGS scores for metrics A and B, that we see that metric B has all its bars on the right half of the plot, while metric A has a very sharp spike on the far right, but a small spike in the left half (still achieving a higher overall average score, but making more severe mistakes). This added detail, which one could not observe through the mere metric rankings, but which is easily noticed by a human observer, might be of interest for several reasons. For instance, a metric developer might judge from these results that although metric A performs well overall, it can be deeply unreliable on occasion, and that the cases where the error is grave might indicate some (albeit minor) conceptual problem with the metric design; while the developer of metric B might take such a distribution to mean that metric B functions well as a broad classifier (gets everything 'vaguely right', makes no large mistakes), and that some mere fine tuning of the metric's mechanisms might be enough to improve its ranking. Additionally, such visual representations aim to help select metrics to use in the creation of hybrid metrics, an approach to improving metrics based on analysis results which we will present in the conclusion.

The second method we propose is another way of comparing any two metrics in more detail. We have the CGS scores for both metrics. For each such pair of DSMs, and for each corpus section, we record the absolute difference in CGS scores of both metrics for each document in that section—we call this the metric divergence score (MD score) which must be a value between 0.0 and 1.0. We then plot the distribution of MD scores and/or compute the average for each section. When two metrics get high average MD scores for a section of the corpus, it means that they are evaluating the type of similarity defining that section in radically different ways (the opposite is evidently true too). This sort of analysis can, again, be useful in several ways. For

[5]Such a distribution can be observed in Figure 6.6c, in Chapter 6.

example, if we know well how metric A works after years of experience, and observe that some new unknown metric performs well on a small testbed corpus, but also has a *low* MD score relative to metric A, then we can posit that it is broadly similar in terms of the properties it exploits to determine similarity, even if the mechanisms by which it does this are different. We can, in turn, use this information to better understand what properties identify that sort of similarity. And again, we will suggest using MD scores to help select metrics for the creation of hybrid DSMs as will be explained in the conclusion.

Chapter 4

Designing a Corpus Construction Module

In this chapter, we will discuss the implementation of the corpus generating framework—`corpusbuilder.py`. As specified in §3.1, the minimum requirement was to create a set of classes with a common interface from which we could extract a section of the testbed corpus. In §4.1 we will present the general basis for such classes. The sections that follow it will describe the specialised corpus builders we used to construct the testbed for our experiment. The actual construction was performed using a simple script—`buildcorpus.py`—shown in §B.2, which we will not go over here. Where they were not too long, we provided samples of corpus entries. A full text version of the corpus is provided with this work on a CD-ROM (*cf.* Appendix A for details).

4.1 Setting Up a General Framework

4.1.1 Structure and Shared Features

We used object-oriented features of Python 2.6 throughout the project, and attempted to write modular code that was easy to extend. The writing of the corpus construction classes, the code for which can be seen in §B.1 of the

appendix, had to begin with a parent class, `corpusbuilder`, that would set the uniform interface features of the specific corpus construction classes, as well as functions and variables that would be shared by the children classes.

A section of the testbed corpus would always be a set of entries with a unique identifier referring to a corpus entry constituted of a triplet, the first member of which is the first document of the document pair, the second member being the second document, and the third member being the gold-standard score for that document pair. The unique identifiers contain specific information. Here is the general structure of such an identifier (segments between rounded parentheses are mandatory, between square brackets are optional):

(CORPUS_SECTION) [-SUBSECTION_TAG] (ID_NUMBER) [m]

For example, an entry for our paraphrase corpus (CORPUS_SECTION = "Paraphrase"), from the MTC subsection (SUBSECTION_TAG = "MTC") could have the unique ID "Paraphrase-MTC2135" (where 2135 is a unique number assigned by the program, as will be explained shortly). Some entry from our abstract-article pair corpus section (CORPUS_SECTION = "Abstract"), which has no subsections, could have the unique ID "Abstract23", as the subsection tag is entirely optional. The last point of interest is the optional trailing 'm'. As was discussed in §3.1.4, it was often possible—and indeed necessary for balanced evaluation—to use pairs from an already constructed corpus to expand it by voluntarily mismatching documents and assigning a gold-standard score of 0.0 to such mismatches. We will discuss the function that does this later, but suffice it to say for now than when corpus entries are produced in this way, we add an 'm' at the end of the ID. Finally, one might ask why all other components of the ID are needed since the ID number is unique (thanks to a global counter). The reason why is that all the extra information contained in the ID will be used to sort documents into sections and subsections and obtain refined results during the analysis phase; we will describe how we do this in Chapter 6. Each corpus builder is therefore given tagging information which contains the corpus type (*e.g.* Paraphrase, Abstract, etc.) as well as a subsection tag stored in the class attribute `tag` (described below) if one was specified. We will not mention the tag details in the descriptions of specific corpus builders in this chapter, but the reader should be aware that they are hard-coded into each class discussed here.

The `corpusbuilder` class has a small initialisation function which declares

some key variables we are about to discuss, and which in turn would need to be called in the initialisation function of specific corpus builder classes.

The key variables declared here were as follows:

- `documents1` and `documents2`: dictionaries that map unique identifiers to strings containing the documents of the document pair in each entry. For example, if the corpus section were to contain an entry with ID 'id1', and with as two documents forming the matched pair a string 'string1' and a string 'string2', then the dictionary `documents1` would have an entry with the key 'id1' mapped to the content 'string1', and the dictionary `documents2` would have an entry with the key 'id2' mapped to the content 'string2'. These dictionaries will be used for the construction of the formatted and annotated corpus section.

- `IDs`: a list (array) containing all the IDs of entries generated by corpus constructing class instances. This is mostly for the sake of convenience (and use within the class), since such information could easily be extracted by reading the keys of `documents1` or `documents2`.

- `corpus`: a dictionary mapping element IDs to corpus entry triplets, as described above.

- `mixedcorpus`: the mixed extension of the corpus, kept in a separate dictionary to cater for the cases where extending the corpus in such a manner is not needed, in which case the program needing the matched corpus section need only read from the `corpus` variable of the corpus building object, and ignore the `mixedcorpus` variable.

- `mixedIDs`: the IDs of entries in the mixed extension of the corpus which, as presented above, end with the tag 'm'.

- `noMix`: a boolean set to 'False' by default. When this is true, any request to extend the corpus by mismatching documents will be ignored. This should be set to 'True' in any corpus constructor that does not require mixing.

- `tag`: a private string variable which will contain the subsection tag (for use in the ID of corpus entries) passed to class instances during their creation in other scripts. Set to an empty string (no tag) by default.

The base `corpusbuilder` class then defines three helper functions for use in

child classes:

- `calcsimscore`: takes a corpus ID as input, retrieves the documents with that ID from `documents1` and `documents2`, and calculates their similarity score. In this parent class, the function simply returns 1.0 as a heuristically-defined gold-standard for matched documents. This function should be overloaded for the implementation of more complex heuristics.

- `scoreCorpus`: pulls all the IDs from `IDs`, and for each one (`ID`), builds a tuple using the documents with that ID from `documents1` and `documents2` and the gold-standard score for those documents as calculated by `calcsimscore(ID)`. It stores such tuples in the class variable `corpus` as the value for a the key `ID`. This function stays unchanged in all children classes, since `calcsimscore` is all that needs to be overloaded to change the scoring heuristic, and leaving this function unchanged ensures homogeneity of corpus structure.

- `generatemixed`: takes the subsection tag (if defined) and an optional integer `quantity` standing for the number of mixed corpus entries to generate (if not specified, it will simply double the size of the corpus under the assumption that all existing document pairs are matched by high degree of similarity). If the class variable `noMix` is True, then this function will do nothing. If not, it randomly mismatches pairs, gives them a gold-standard score of 0, and adds them to `mixedcorpus` with a unique ID, which it also adds to `mixedIDs`. Because of `noMixed`, this function can be safely called in any subclass initialisation function after `scoreCorpus`, whether the subclass needs to generate a mixed corpus or not.

To summarise what child classes must do in their initialisation function in order to generate their corpora:

1. Extract document pairs from source corpora and write them to `documents1` and `documents2` with a unique ID which must be added to `IDs`.

2. Optionally provide a scoring heuristic using the structure of the source corpora by overloading `calcsimscore`, or simply use the existing naïve heuristic.

3. Call `scoreCorpus` and `generatemixed`.

After an instance of such a subclass is created, the script needing the matched corpus can simply copy the `corpus` attribute of that instance to some local variable. If it needs a full corpus section, it can merge the `mixedcorpus` attribute of that instance into the dictionary to which it copied the `corpus` attribute.

4.1.2 Suggested Improvements

We suggest no particular functional improvements to this general class, as it worked perfectly for the purposes of our experiment. But since the subsidiary aim of this project is to create an *easily* extendable framework, we recognise that the code of this module could use some refactoring in order to further facilitate the writing of new child classes. However, since the way each subclass of `corpusbuilder` extracted their documents was widely different, this would require some heavy modification of the subclasses we had already written. Deadlines and results were more important in the case of our experiment than the aesthetics of the corpus generating classes, so we leave it to those interested in extending this module to decide if it is worth rewriting the base class.

4.2 Syntactic Toy-Corpora

4.2.1 General Overview

The two corpus constructing subclasses of `corpusbuilder` presented in this section—namely `editcorp` and `POSswitchcorp`—are designed to generate testbed corpus sections where document pairs are related by principally syntactic features. The two interesting characteristics these corpus builders present, relative to the others, is that the notion of similarity with which they are constructing the corpus is well defined, and that this similarity is exactly quantifiable during the construction process; this is to say, we can set a gold-standard for document pairs which is an objective measure of similarity, rather than a score obtained through some heuristic.

The reason why such precision is possible is that the notions of similarity

at play in both cases are very simplistic, and that some of the documents generated by these classes are fairly unusual instances of language (or in fact completely garbled). This is because the reason we included these corpus sections is to have some basis for demonstrating what sort of similarities with which principally syntactic metrics were designed to work bes. Because of this we will refer to these corpora as 'toy-corpora'.

4.2.2 Construction Process: Random Edits Corpus

The first of these toy-corpora is `editcorp`, a corpus which pairs each sentence with a version of itself which has undergone a randomly determined number of edits. An edit, within this context, refers to the change of a single character according to a set of operations: morph, delete, move. These correspond to class functions which we will describe shortly.

The number of edits used to generate the second sentence of the pair from the first is what determines the gold-standard according to the following formula:

$$\text{Gold-Standard} = 1.0 - \frac{\text{\# of edits}}{\text{Sentence Length}}$$

The sentence length referred to is the length of the original sentence (since the edited sentence may be shorter), the number of edits is randomly set for each corpus entry as will be discussed below. When the number of edits tends towards the sentence length, we consider that significant syntactic change has occurred; correspondingly, $\frac{\text{\# of edits}}{\text{Sentence Length}}$ tends towards 1.0, hence the gold-standard score tends towards 0.0, and conversely when there are few edits relative to the length of the original sentence, $\frac{\text{\# of edits}}{\text{Sentence Length}}$ tends towards 0.0, hence the gold-standard score tends towards 1.0.

According to good object-oriented design rules, we should have implemented this measure in `calcsimscore`, overloading the eponymic function from the parent class `corpusbuilder` and then calling `scoreCorpus`, however because of the simplicity of this class we found it simpler to process the same operations as part of the initialisation function. The initialisation function works as follows. We begin by importing a set of 500 raw (untagged) sentences[1] from

[1] This number can be changed for other experiments by specifying the size of the corpus as a keyword argument when creating an instance of `editcorp`, *e.g.* by writing `myeditcorp = editcorp(size = 1000)`, we would build a 1000 entry corpus.

the Brown Corpus[2]. We obtained the pre-chunked sentences from the Brown corpus module integrated into the Natural Language Toolkit (NLTK)[3]. We did not use any of the part-of-speech annotations or the categorisation feature, and a similar corpus generator could be written using any corpus with raw text as a basis (one would only need to chunk out the sentences). We could well have written a more general class to make doing this easier, but since this class is written to generate a toy-corpus, we did not judge this to be worthwhile.

Each sentence in the thus-obtained set of sentences is an ordered list of tokens (words). For each such list, we first join it into a string, then use a class function `randomedits` (described below) to obtain a string with a random number of edits, as well as the number of edits performed to obtain it. Using this number and the formula discussed above (Gold-Standard $= 1.0 - \frac{\#\text{ of edits}}{\text{Sentence Length}}$), we obtain the gold-standard score; we finish by writing the original sentence, edited sentence and gold-standard to the class's `corpus` dictionary with a new ID as key. We perform this task for all other sentences in the set obtained from the Brown Corpus in order to produce our full corpus section.

The `randomedits` function discussed above is the cornerstone of this subclass. It takes the original string as an argument which it copies to a variable restricted to the function's scope (*i.e.* deleted when the function returns a value), randomly decides on a number of changes between 0 and the length of the original string, and then it performs the following set of steps as many times as the randomly decided number of changes. First, a random number generator selects a location on the string to be edited, then it randomly selects one of the three edit operations[4]. If the `move` edit (described below)

[2]For more details on the Brown Corpus, a POS-tagged and categorised collection text, see (Francis and Kucera 1979), available online at `http://khnt.aksis.uib.no/icame/manuals/brown/`.

[3]We made fairly frequent use of NLTK throughout this project. It was initially developed by (Bird and Loper 2002), and is presented in detail in (Bird et al. 2009). It is open source and available online at `http://www.nltk.org`. We will describe the other functions and features of the toolkit that we used as they appear in our implementation account. Although this toolkit provides implementations of entire DSMs, it will only be used in a small number of lines in the source code presented in Appendix B, since most of the framework code is used to set up DSM evaluation.

[4]There is a non-zero possibility that the random number generator will select the same character for the same edit function twice in a row, and that the random edit will effectively

is selected, another random edit location (the move's destination) is selected in the string. Next the edit operation is performed, modifying the string. Finally a counter containing the number of edits is incremented by one. This is repeated as many times as the decided number of changes, and finally, the modified string and number of changes are packed into a 2-tuple which is returned.

The edit functions used were as follows:

- `morph`: takes a string and edit location as arguments, randomly selects a character from a restricted set ASCII characters[5], and checks if the selected character is the same as the character at the string's edit location. If the character is the same, it selects another one and performs the same check, if it is not, it replaces the old character with the new and returns the string.
- `delete`: takes a string and edit location as arguments, returns the string with the character at the edit location removed (shortening the string's length by 1) by concatenating the string segment before the edit location with the segment after the edit location.
- `move`: takes a string, edit location, and destination as arguments, copies the character at the edit location into a buffer, non-destructively inserts the buffer contents at the destination by splicing and re-joining the string at that location on either side of the added material, and finally removes the character at the edit location by calling `delete` and returns the edited string.

To finish describing this sub-class, one may ask why there is no mixed corpus. Because the random number generator has a uniform distribution, over a large number of corpus entries the gold-standard scores should cover the range between 0.0 and 1.0 and average to 0.5. Therefore mixing is not needed, as it would skew the distribution of gold-standard scores. We set the attribute `noMix` to 'True', which is solely for external use since the initialisation function doesn't call `generatemixed` (but even if it did, nothing would happen since `noMix` is true).

revert the string to a state it was in a few steps before without decreasing the edit counter. This will indeed skew the gold-standard score for this entry, but the low probability of this occuring and the large size of the corpus renders such an issue negligible.

[5]Character set: `1234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ,.;:?!`

A sample of this section is the following two entries:

```
<doc1 ID=Edits100> -- Committee approval of Gov.  Price
Daniel's `` abandoned property '' act seemed certain Thursday
despite the adamant protests of Texas bankers .  </doc1>
<doc2 ID=Edits100> -c o tteLapgo XfGv.PTe DSlr'g6e dgcanovprekreyyrS1KNtThmJd
caRSn3ysCFSetpet ehiamvnom pestsVdoeah n 7 </doc2>
<score ID=Edits100> 0.179310344828 </score>

<doc1 ID=Edits10> The City Purchasing Department , the
jury said , `` is lacking in experienced clerical personnel
as a result of city personnel policies '' .  </doc1>
<doc2 ID=Edits10> The City Purchasing Department p, the
jury said , ``is lacking in exerienced clerical personnel
as a result of city personnel policies '' .  </doc2>
<score ID=Edits10> 0.985714285714 </score>
```

4.2.3 Construction Process: POS-Tag Equivalence Corpus

The second toy-corpus builder is `POSswitchcorp`, which creates document pairs where the first document is a sentence obtained from the Brown Corpus, and the second is an edited version of the first where a random number of words have been substituted for other words with an identical part-of-speech (POS) tag. The notion of similarity at play is that of superficial similarity between grammatical structures. Two sentences are said to be the same if they have the same grammatical structure, and different if they do not. A more sophisticated corpus builder could be designed to offer a more complex notion of similarity involving parts of speech, but we believed this toy-corpus was enough to produce some interesting initial results.

The class contains only an initialisation function, and calls helper functions from the parent class `corpusbuilder`. The initialisation function works as follows: first, a set number[6] of tagged sentences are extracted from the Brown Corpus using the NLTK interface, and stored in `tagged_sents`. We then

[6] As was the case for `editcorp`, the default value can be changed to use different parts of the corpus by giving the initialisation argument `startindex=`[start position]; and the section size can be modified by providing the keyword argument `size=`[section size].

create a dictionary `POScatwords` which will map POS-tags to word lists. We populate this dictionary using the sentences which we extracted (for each word/POS pair in the sentence we add that word to the list that the POS-tag maps to in `POScatwords`).

Next, for each sentence in `tagged_sents`, a random number of switches are selected. For each switch in the range selected, a word is chosen at random in the sentence being edited, and the list of words for that POS is retrieved from `POScatwords`. A word is chosen at random and substituted for the original. This operation is repeated as many times as decided, and the final edited sentence is paired with the original sentence, packed into a triplet with a gold-standard score of one, and added to the class dictionary `corpus` under a newly generated ID by calling the class function `scoreCorpus`. We do not need to modify the naïve heuristic in the parent class function `calcsimscore`, since the sentences generated this way must have the same grammatical structure as the originals (even if the generated sentences may seem a bit unnatural or nonsensical at times).

Finally, we call `generatemixed` to generate the mixed corpus, as described in §4.1.1.

A sample of this section is the following two entries:

```
<doc1 ID=POSswitch1000> In 1961 , it is estimated that
multiple unit dwellings will account for nearly 30 per
cent of the starts in residential construction .  </doc1>
<doc2 ID=POSswitch1000> In 1961 , she is estimated little
multiple staff dwellings will account for nearly 30 per
cent across the starts in residential color ; </doc2>
<score ID=POSswitch1000> 1.0 </score>

<doc1 ID=POSswitch1004m> Mr.  and Mrs.  B. Lewis Kaufnabb
, for senior aides , and Mrs.  Samuel P. Weinberg , for
the bundles .  </doc1>
<doc2 ID=POSswitch1004m> ‘‘ And They bum tickets to everything
We can ’’ , she said .  </doc2>
<score ID=POSswitch1004m> 0.0 </score>
```

4.3 Theme-grouped Texts

4.3.1 General Overview

The corpus builder `themecorp` pairs documents—short texts a few paragraphs long—by theme (adventure, editorial, romance, religion, science fiction). The pairs within each theme are meant to have loose lexical relations, in that the key words in document pairs may not have a direct, tight lexical relation to each other (as the words 'spoon' and 'fork', or synonyms would) but are generally of the same broad lexical family: there is a particular kind of language common to most romance texts that is far different from that found in science fiction.

4.3.2 Construction Process

The class `themecorp` is a subclass of `corpusbuilder`. However, not unlike `editcorp` and `POSswitchcorp` described in §4.2, it does the work its parent's `scoreCorpus` and `calcsimscore` functions would normally take care of directly in its short initialisation function, due to its relative simplicity.

This class has two special attributes, `cats` and `dualcats`. First, `cats` is a list of five Brown corpus categories/genres which we arbitrarily picked. Through a bit of indirection, this could be made into the sort of thing users would pass to the class during initialisation of instances, but we opted for a quick implementation to get results. Second, `dualcats` is dictionary mapping each category from `cats` to its *dual category*. For each category, the dual category is a category from *cats* which we decided was most *unlike* the first (*e.g.*). This will be required for creating the mixed corpus, as should become evident later.

All the work for this function is done in the initialisation function. For each category in `cats`, news 'chunks' of text are extracted from the Brown corpus using the NLTK interface by joining together three successive new paragraphs. The text chunk is then added to a dictionary entitled `text_chunks` mapping categories to lists of such chunks. This operation is performed fifty times (this number, and the size of chunks are default values, which can be changed by the user when instantiating the object).

Next, for each category in `cats`, document pairs are formed by taking consecutive chunks from the category to form document pairs. Such pairs are packed into a tuple with a gold-standard score of 1.0—as the documents share the same theme—and added to the class dictionary `corpus` under a fresh ID. This is repeated for each category to form the matched part of the testbed corpus section. Next we create the mixed part through a similar process. For each category, each document is paired up with a document from the dual category, found using the dictionary `dualcats`. Such mixed pairs are then assigned a gold-standard score of 0.0 (since they are documents from different thematic categories) when packed into a tuple, which is then written to the class dictionary `mixedcorpus` with a fresh mixed ID. This is repeated for each document in each category to form the mixed corpus.

Here is a sample entry from this section:

```
<doc1 ID=Themes1501> Dan Morgan told himself he would
forget Ann Turner .  He was well rid of her .  He certainly
didn't want a wife who was fickle as Ann .  If he had married
her , he'd have been asking for trouble .But all of this
was rationalization .  Sometimes he woke up in the middle
of the night thinking of Ann , and then could not get back
to sleep .  His plans and dreams had revolved around her
so much and for so long that now he felt as if he had nothing
.  The easiest thing would be to sell out to Al Budd and
leave the country , but there was a stubborn streak in
him that wouldn't allow it .The best antidote for the bitterness
and disappointment that poisoned him was hard work .  He
found that if he was tired enough at night , he went to
sleep simply because he was too exhausted to stay awake
.  Each day he found himself thinking less often of Ann
; ; each day the hurt was a little duller , a little less
poignant .  </doc1>
<doc2 ID=Themes1501> He had plenty of work to do .  Because
the summer was unusually dry and hot , the spring produced
a smaller stream than in ordinary years .  The grass in
the meadows came fast , now that the warm weather was here
.  He could not afford to lose a drop of the precious water
, so he spent most of his waking hours along the ditches
in his meadows .He had no idea how much time Budd would
```

```
give him .  In any case , he had no intention of being
caught asleep , so he carried his revolver in its holster
on his hip and he took his Winchester with him and leaned
it against the fence .  He stopped every few minutes and
leaned on his shovel as he studied the horizon , but nothing
happened , each day dragging out with monotonous calm .When
, in late afternoon on the last day in June , he saw two
people top the ridge to the south and walk toward the house
, he quit work immediately and strode to his rifle .  It
could be some kind of trick Budd had thought up .  No one
walked in this country , least of all Ed Dow or Dutch Renfro
or any of the rest of the Bar B crew .  Morgan watched
the two figures for a time , puzzled .  When they were
closer and he saw that one was a woman , he was more puzzled
than ever .  </doc2>
<score ID=Themes1501> 1.0 </score>
```

4.4 Wikipedia Article Pairs

4.4.1 General Overview

The corpus builder `wikicorp` pairs up Wikipedia articles (one from English Wikipedia, the other from Simple English Wikipedia[7]) on the same specific subject, considering this to be a broad form of lexical similarity which is different from that obtained from `themecorp` (*cf.* §4.3) since the document pairs obtained in `wikicorp` will contain key words with closer lexical relations (same subject) than in `themecorp` (same theme).

4.4.2 Background and Sources

For our experiment, we collected 122 English article URLs which we put into a file. We wanted articles which had a Wikipedia Simple English version, and obtained them through the following methodology. We went to Wikipedia

[7]See §3.1.1 for an explanation of Simple English Wikipedia.

Simple English, and repeatedly used the random article link (http://simple.wikipedia.org/
to cycle through pages. When pages met the criteria discussed below, the URLs were saved. When enough URLs were obtained, a quick search and replace would substitute en.wikipedia.org for simple.wikipedia.org to obtain a list of English Wikipedia article URLs.

The selection criteria were fairly simple. First, we ignored all articles that were very short (one or two sentences), since there was very little basis for lexical comparison. Second, we avoided articles that had large amounts of tabulated data or links and otherwise little article text, as the tabulated data was generally copy-pasted from the English article, and thus the similarity would be too trivial.

We recognise that the selection methodology could be perfected, but the URL lists are not hard coded. Since the class can retrieve article pairs intelligently enough to check if there exists a Simple English version for an English article, one could feasibly put the URLs of all the English Wikipedia articles into a list and obtain a corpus from that, so the class allows for further work without the need for any code modifications.

4.4.3 Construction Process

The class wikicorp is a subclass of corpusbuilder. Its initialisation function takes the address of a file where URLs of English Wikipedia articles are listed (one URL per line), to be passed during creation of class instances. For each URL in this list, the URL is changed to point to the 'printable' version of the article (with more uniform formatting) if needed, by using the class function getprintURL which use a simple regex to add &printable=yes to a slightly reformatted URL.

The properly formatted URL is then passed to the class function getHTML, which uses Python's url-opening library—urllib2—to present the script as a Mozilla browser (otherwise Wikipedia restricts access) and obtain the raw HTML from the given URL, which it then returns to the initialisation function.

A regular expression is then used to locate the Simple English version of the English article the raw HTML of which has been retrieved (principally by searching for simple.wikipedia.org), which is then used to retrieve the raw

HTML for the Simple English version of the article. The raw text from both articles is then obtained by passing the raw HTML to a NLTK helper function `clean_html` which will strip the HTML tags and formatting commands from any string passed to it, and the text is stored in local variables `text_eng` and `text_simp` (for the English and Simple English articles, respectively).

Finally, the text of the articles is put through the class function `getarticlebody` which uses features such as the text 'Retrieved from "`http:/`' to determine the beginning of boilerplate text present in every article and delete it. It also cleans up the superfluous whitespace present in the article, although this is more useful for humans wanting to consult the corpus than for DSM evaluation, since most metrics will just ignore whitespace. To complete the journey, the cleaned text of the English article and Simple English article is added to the class dictionaries `documents1` and `documents2` under a fresh ID.

The above steps are repeated for every URL in the list. After this, the parent class function `scoreCorpus` is called to assemble the corpus from the pairs obtained from `documents1` and `documents1` and using the default heuristic in `calcsimscore` (from the parent class) of assigning a gold-standard score of 1.0 to each pair (since they are matched). The corpus is thus written to the class dictionary `corpus`. To complete corpus construction, `generatemixed` is called to write the mixed segment of the corpus to `mixedcorpus`.

4.4.4 Future Improvements

We believe this class works the best it could work. The regexes used for cleaning up the raw text could be fine tuned, but this is principally an aesthetic change. The principal improvement those considering doing further work with this class should consider is to do with collecting URLs, as briefly discussed in §4.4.2: one might wish to use the subcategory feature all the classes in this module have to divide the source documents by category of article (people, places, concepts, etc.) and see if the performance of similarity metrics within these subcategories varies.

4.5 Paraphrase Corpora

4.5.1 General Overview

The class `lapatacorp`—perhaps the most complex class in the module—serves to build paraphrase sections of the test-bed corpora, where in each document pair is associated by semantic similarity: one document is a sentence and the other one is its paraphrase, obtained through different methods for each subsection. This corpus constructor can be used for any further work where the sources are formatted and annotated as was done in the work by (Cohn et al. 2008). We will describe the background an implementation of this class in this section.

4.5.2 Background and Sources

While looking for corpus sources for various sorts of paraphrase, we came across work by (Cohn et al. 2008) on M. Lapata's site (hence the class name), which provided paraphrase pairs of three different types: one set was obtained from the Microsoft Paraphrase Corpus designed by (Dolan et al. 2004), and two sets were different translations of sentences, one from Chinese news articles, the other from Jules Verne's *Twenty Thousand Leagues Under the Sea* (originally in French). We decided that last two deserved different classes, since the structure of French sentences is relatively close to that of English sentences, while Chinese is fairly different, and that as a result there might be a different degree of syntactic variation between the pairs of both sources.

In addition to providing us with pre-paired source documents, (Cohn et al. 2008) had produced alignment tables for each paraphrase pair, and then had human annotators (named 'A' and 'C') view the tables and judge whether particular alignments were good, average or bad (the specific annotations used were 'Sure' (S), 'Possible' (P) and '-' when no alignment was found).

For each of the three source categories, there are four significant files (here `zzz` is the name of the category, namely 'mtc', 'news' or 'novels'):

- `zzz.common.one` and `zzz.common.two`: these are the *document source files* which contain the paraphrase sentences alongside a unique ID. We can construct paraphrase pairs by taking a sentence with some ID

from zzz.common.one and pairing it with the sentence with the same ID from zzz.common.two.

- zzzAx.align and zzzCx.align: these are the *annotation source files* which contain the paraphrase alignment quality annotations for annotators A and C, respectively. They (bizarrely) are not given the same IDs as in zzz.common.one and zzz.common.two, but the order is the same such that the first sentence being annotated is the first sentence to appear in zzz.common.one and zzz.common.two, so the proper IDs can be recovered.

Below, we will explain how we used these annotations to derive a more sophisticated scoring heuristic for this corpus section while discussing the implementation of this class.

4.5.3 Construction Process

The class lapatacorp is a subclass of corpusbuilder. It has a set of dictionaries which we will populate using the source corpora:

- tokens1 and tokens2: dictionaries mapping unique IDs to list of tokens such that the tokens under some ID in tokens1 form a sentence which is the paraphrase of the sentence formed by the tokens in tokens2.
- annotations_A and annotations_C: dictionaries mapping unique IDs to lists of annotations ('S','P' or '-') such that for any ID, the elements of the list of annotations in annotations_A and annotations_C map onto the lists of tokens under the same ID in tokens1 and tokens2, respectively.

The initialisation function takes four filenames for zzz.common.one, zzz.common.two, zzzAx
.align and zzzCx.align (for one of the three classes we have, namely 'mtc', 'news' and 'novels', or any corpora with the same structure) while instantiating the class and stores them in class variables rawfile1, rawfile2, fileA and fileC, respectively.

It first passes rawfile1 and rawfile2 to the class function getdocs, which collects all the document pairs (in the form of lists of tokens) from these files using class function rawparser and assigns each pair a new ID using

the appropriate tags (*cf.* §4.1.1). The function `getdocs` also reconstructs the sentence strings by joining the tokens together. It returns the sentence strings, which are put into class dictionaries `documents1` and `documents2`, as well as the tokens which are put into class dictionaries `tokens1` and `tokens2`, and also the list of generated IDs, put into class list `IDs`. The function `rawparser` that was used here uses regexes to match sentences from `rawfile1` and `rawfile2` based on the XML-like tags (Cohn et al. 2008) have used to encode IDs.

Next the initialisation function passes `fileA` and `fileC` to class function `annotparser` to populate `annotations_A` and `annotations_C`. This function checks the order of class list `IDs` to match annotation sequences to the IDs of sentences we extracted using `getdocs`. Each line in the annotation file contains:

1. an ID (used in conjunction with class list `IDs` to assign an ID entry for each set of annotations),
2. the position of a word from `tokens1[ID]`,
3. the position of a word from `tokens2[ID]`,
4. an alignment annotation: '`S`' for sure (good alignment), '`P`' for possible alignment, and '`-`' for non-applicable/bad alignment.

For each ID, the annotations from `fileA` and `fileC` are copied into dictionaries mapping IDs to lists of annotations, which are then returned and stored in `annotations_A` and `annotations_C`.

Following this, the parent class function `scoreCorpus` is called to compile tuples of documents pairs and gold-standard scores, and store them—using the document IDs as keys—in the class dictionary `corpus`. The function operates as it does in `corpusbuilder`, however the function it calls—`calcsimscore`—is overloaded. The local version of `calcsimscore` simply takes the *F-measure of inter-annotator agreement* to be the gold-standard score for a document pair. This is because annotators are likely to say that word alignments are good if words are aligned with other words that have similar meaning/use. We calculate the F-measure by taking the annotations of one annotator to be the reference annotations and the annotations of the other to be test annotations. We call As the set of cases where alignments are given the annotation '`S`' by annotator A, Ap when they are given annotation '`P`', Cs when they

are given annotation 'S' by annotator C, and Cp when they are given annotation 'P'. We thus obtain precision and recall for the alignments of a pair of documents as follows (for a set s, $|s|$ is the size of s):

$$\text{Precision} = \frac{|As \cap Cp|}{|As|} \qquad \text{Recall} = \frac{|Ap \cap Cs|}{|Cs|}$$

The F-measure or F-score is obtained through the formula:

$$\text{F-score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F-score is returned to `scoreCorpus` by `calcsimscore` for every document pair and sets the gold-standard score for that pair.

To finish, the parent class function `generatemixed` is called to form the mixed corpus. We can 'safely' call this because almost all the pairs have good alignments, and thus without mixing, the average gold-standard score is just below 0.9. Ideally, we could determine how many documents we need to add to the mixed corpus to bring the overall average down to 0.5, but for the sake of simplicity (since the difference was not much for our source corpora), we simply left the default setting of doubling the corpus size here. The generated mixed corpus is written to class variable `mixedcorpus`, which concludes the running of the initialisation function.

Here are three sample entries from this section:

```
<doc1 ID=Paraphrase-MTC2001> it was expected that these
two men became in-laws while they were in jail .  </doc1>
<doc2 ID=Paraphrase-MTC2001> it is quite out of expectation
that the two men became relatives by marriage when they
are imprisoned in the north .  </doc2>
<score ID=Paraphrase-MTC2001> 0.833333333333 </score>

<doc1 ID=Paraphrase-News2201> she has also signed a contract
with random house to write two books .  </doc1>
<doc2 ID=Paraphrase-News2201> pauley also announced thursday
that she has struck a deal with random house to pen two
books .  </doc2>
<score ID=Paraphrase-News2201> 1.0 </score>

<doc1 ID=Paraphrase-Novel2461> ‘‘ the only difficulty
, ’’ continued captain nemo , ‘‘ is that of remaining several
```

```
days without renewing our provision of air . '' </doc1>
<doc2 ID=Paraphrase-Novel2461> `` our sole difficulty
, '' captain nemo went on , `` lies in our staying submerged
for several days without renewing our air supply . ''
</doc2>
<score ID=Paraphrase-Novel2461> 0.777777777778 </score>
```

4.5.4 Future Improvements

Aside from the suggested further improvement regarding the balance-motivated way of generating the mixed corpus, there are few things to improve upon here, we believe. This class is general enough to cater to the creation of corpora from any source documents the user wishes to supply, as long as they are formatted in the same way (Cohn et al. 2008) formatted theirs. Researchers wishing to pursue further work with this class may therefore want to consider increasing the size of the corpus through the same alignment and annotation process as discussed by (Cohn et al. 2008), perhaps opting to also use purposefully misaligned pairs, so that we could add more granularity to lower scoring document pairs and avoid using `generatemixed` (which uses a fairly naïve heuristic).

4.6 Abstract-Paper Pairs

4.6.1 General Overview

The final corpus builder class implemented is `papercorp`, which matches academic articles with their abstracts on the grounds of semantic-lexical similarity (summarisation). Since abstracts are provided with the articles in papers, this class is mostly about using regexes to separate the abstracts from the article.

4.6.2 Background and Sources

We obtained a set of 187 articles from the website of the Association for Computational Linguistics (specifically from `http://aclweb.org/anthology-new/P/P08/`). As discussed in §3.1.1, we chose all the papers to be on the same class of topics (in this case, computational linguistics) in order to weaken lexical criteria for matching, focussing more on the semantic aspect of summarisation. We leave the mixing of sources to future work, since this class (like all others in the module) has full support for custom sources and sub-categories (passed to the object through the `tag=`*subcategory-name* during class instantiation).

4.6.3 Construction Process

The class `papercorp` is a subclass of `corpusbuilder`. Its initialisation function takes as argument a folder where the PDF files of the articles from which it is to construct the testbed section are stored (*NB.* it will use all PDFs in this folder).

It first gets the absolute path of the folder address passed to it (and complains if it does not exist). We then use Python's pathname pattern expansion library—`glob`—to get a list of all the PDF filenames in the user-specified folder, which are saved to the list `pdflist`. For each PDF filename in `pdflist`, the function will attempt to get the raw text by passing the PDF filename to class function `getrawtxt`. If anything fails during the running of `getrawtxt`, the program will notify the user via `stderr` and continue onto the next PDF in `pdflist`. The function `getrawtxt` takes a PDF filename, and attempts to extract the raw text as follows: it will first generate a new text filename based on the PDF's filename—`pdf`—which it save to the local string `textFN`. It will use the standard Python `call` function to run the bash command line command "/usr/local/bin/pdftotext -nopgbrk -raw [`pdf`] [`textFN`]". This calls the program `pdftotext` from the open-source Poppler PDF rendering library[8] which is instructed to read the file `pdf` and write the raw text within to the text file `textFN`. The contents of the raw text version of the PDF with filename `textFN` are then read into a string stored in variable `raw`, which is returned to the initialisation function, which adds the thus-obtained string to the list `rawpdfs`.

[8]See `http://poppler.freedesktop.org/` for more details.

Once the above has been repeated for all PDFs in `pdflist`, we must separate the abstract from the body, which we do as follows. For each raw article string in `rawpdfs`, we obtain the abstract and body from that article by passing the string to class function `chopabstract`. If it is unable to do so, it will report this to `stderr` and continue onto the next article string. The function `chopabstract` works as follows: first it will look for the word 'introduction' in the first 2500 words of the article. If it does not find it, it will look for the word 'keywords' in the same range. If it does not find *that*, it will raise an error, and the article string will be skipped. If it finds either of these (in the order they were checked), it considers it to be the beginning of the article. If the word 'abstract' is used at the beginning of the abstract, it will treat that as the beginning of the abstract. If not, it will treat everything above the introduction (including headers and author information) as part of the abstract. Using regexes and the aforementioned cut-off points, it will separate the abstract and article body and pass both separately to the class function `cleantext` which removes superfluous whitespace using regexes, before finally storing them into local variables `abstract` and `body`, respectively. Finally, it returns the abstract and article body to the initialisation function, which in turn adds the abstract to class dictionary `documents1` under a fresh ID, and the body to `documents2` under the same ID.

Once this is done for all raw texts, the parent class function `scoreCorpus` is called to build the corpus using the parent's `calcsimscore` function with its naïve heuristic of assigning 1.0 as gold-standard to all matched pairs, storing the thus-produced corpus in class attribute `corpus`. It finally calls `generatemixed` to generate the mixed extension of the corpus.

4.6.4 Future Improvements

The key improvement that could be made here is the heuristic used to determine the beginning of the article in the function `chopabstract`. We relied heavily in the use of the word `introduction` and/or `keywords` because these are common in many computer science papers. However some papers begin with a roman numeral heading the introduction, or don't even have an abstract. The method we used to extract raw text makes it difficult to use formatting aspects of the PDF and spacing, since they are lost in the conversion to raw text, and completing the task using the PDF's formatting was

beyond our technical capabilities, but if a more sophisticated 'chopper' was to be designed, this would be the way to go.

Chapter 5

Evaluating Document Similarity Measures

In this chapter, we will discuss the implementation of the metric evaluation framework—`corpusevaluation.py`. As specified in §3.2, the minimum requirement was to create a set of classes which, when instantiated and given a corpus produced by the classes described in Chapter 4 as argument, will produce evaluation data—for each corpus section and for each DSM—which is suitable for analysis by the script described in Chapter 6. We will also present the theoretical background for these metrics, the tools involved in the construction of the classes, and the metric score normalisation steps taken, as discussed in §3.2.2. The code for all these modules can be found in §§B.4–B.5.[1] The actual evaluation is not run within the framework, but by a separate simple script—`evaluatecorpus.py`—which is shown in §B.6 but not described here.

[1]One metric implemented but not described here is a random evaluator which gives each corpus pair a random score between 0 and 1. This is used to determine what 'better than random' scoring is for each corpus section if the average gold-standard score is not 0.5, and also to show what ineffective DSMs look like graphically when we break down the scores as described in §3.3.2.

5.1 Generalised Evaluation Framework

As we did for our corpus construction framework, we used object-oriented design methods to create an easily extensible framework for evaluating metrics. We used inheritance more rigourously and wrote a large base class, so that all that was needed to implement most metrics was to create subclasses that overload a single function. In this section, we shall present the base class `evaluator`.

The class `evaluator` serves as common ancestor to all classes in this module, although some may have more direct parents as we shall see in §5.4. The class is given as initialisation parameters either a keyword `corpus` pointing to a corpus object produced by the construction metrics detailed in Chapter 4 (namely a dictionary mapping IDs containing section data to triplets constituted of two paired documents and their gold-standard similarity score), *or* it will receive pre-processed data provided by the keyword `eval_data` pointing to a formatted corpus object, as can be created by the pre-processing script—`prepcorpus.py`[2]. If both keywords are specified, it will consider the contents of `corpus` to be more important and ignore `eval_data`.

If `corpus` has been passed to the initialisation function, it is formatted and saved to `eval_data` by passing it to the class function `format`. The function `format` standardises the corpus strings by tokenising them using NLTK's sophisticated tokeniser (capable of, for example, recognising that acronyms and words such as 'Mr.' are tokens), which are then rejoined to form the normalised document strings[3]. The tokens are preserved during this operation. Finally, each corpus entry—which previously was of the format '(doc 1, doc 2, gold-standard score)'—is repacked into the 5-tuple '(normalised doc 1, normalised doc 2, tokens of doc 1, tokens of doc 2, gold-standard score)' which is stored in a dictionary under the same ID as the original corpus entry. These formatting steps are repeated for all original corpus entries. The thus-produced dictionary is then returned to the initialisation function and stored in the class dictionary `eval_data`. If, instead, `eval_data` was passed

[2] The script `prepcorpus.py` is shown in §B.3 but not described here, since it uses the same formatting function `format` as described in this section and the same tagging function `tagcorpus` as described when discussing the intermediary class `TAGevaluator` in §5.4.

[3] For instance, "Mr. Johnson, Bob, and I went to the fair. It was lovely." would be normalised as "Mr. Johnson , Bob , and I went to the fair . It was lovely ."

to the initialisation function, then no formatting is needed, `format` is not called and the dictionary is simply copied to an eponymous class dictionary `eval_data`.

The next and final step for the initialisation function is to call the function `getscore`, which will return a dictionary mapping document IDs to metric CGS scores, as discussed in §3.2.1. The function `getscore` works as follows: for every ID key in the class dictionary `eval_data`, the gold-standard is extracted from the corresponding formatted corpus entry. The corpus entry is then passed to class function `scoreline` which returns the metric score for the document pair, which is stored in the variable `linescore`. The CGS score is then calculated by taking 1.0 minus the absolute difference between the gold-standard score and the metric score for that entry, and it is saved to the dictionary `scores` which maps corpus IDs to CGS scores. This is repeated for every corpus entry, and finally the dictionary `scores` is passed back to the initialisation function, where it is stored in the class dictionary `scores`.

The final piece of the puzzle is the function `scoreline`. If this function is actually called for the `evaluator` class, it raises an exception, since the base class is not meant to stand for any metric and cannot, therefore, score corpus entries. This is the function which is overloaded—sometimes with very simple functions—while implementing the metric evaluation classes, as we will now show.

5.2 Implementing Purely Syntactic Metrics

We collected a selection of syntactic metrics commonly-referred to in the literature, and produced a set of classes to implement them. In this section, we present them, their theoretical background, tools we used to implement them, and how we adapted the class function `scoreline` to complete the evaluation class.

5.2.1 Character-Count and Word-Count

The first two such classes are `charcounteval` and `wordcounteval`, subclasses of `evaluator` which implement a metrics judging the similarity of documents according to character-count and word-count, respectively, such that two documents will have high similarity scores if the character/word-count of one is close to that of other relative to the size of the larger string. This metric score is normalised since the absolute difference between the character/word-counts will always be inferior to the character/word-count of the largest of two document strings, since there are no negative lengths. We subtract this ratio from 1.0 to obtain a still-normalised metric score that tends towards 1.0 when the character/word-count of one document tends towards that of the other, and tends towards 0.0 when the character/word-counts diverge.

The class function `scoreline` in `charcounteval` overloads that of `evaluator`. It takes a corpus entry, extracts the document strings and divides the difference between their lengths (hence the difference between their character-counts) by the largest of the two lengths and subtracts this from 1.0, thus obtaining the metric score for the corpus entry, which it returns to calling function.

The class function `scoreline` in `wordcounteval` overloads that of `evaluator`. It takes a corpus entry, extracts the document tokens and divides the difference between the length of each token list (hence the difference between their document word lengths since each token is a document word) by the largest of the two lengths, thus obtaining the metric score for the corpus entry, which it returns to calling function.

5.2.2 Levenshtein Edit Distance

The class `leveneval` is a subclass of `evaluator` and implements the Levenshtein edit distance metric first introduced by (Levenshtein 1966). The Levenshtein edit distance from one document to another is the *minimum* number of character edits needed to transform one document to another, where the allowed edits are inserting a character, changing a character to some other character, and deleting a character (each count as one edit). This may seem fairly similar to our `editcorp` class discussed in §4.2.2, being based on the same edit principles.

To implement the metric, we overload `scoreline` by writing a class version which takes a corpus entry, extracts the document strings, finds the maximum character length of the two strings and records it to `maxlength`. It then calculates the Levenshtein edit distance between the two strings by passing both of them to the function `distance`, which returns the edit distance. The function `distance`, like a few other syntactic metrics used in this project, was provided by the third-party Levenshtein Python library[4]. We divide by the maximum string length for the same normalisation reasons as in `charcounteval`: the *minimum* number of edits cannot exceed the length of the largest string. The result of the division is then subtracted from 1.0 to obtain the metric score for that corpus entry, which is returned to the calling function.

5.2.3 Jaro-Winkler Distance

The class `jaroeval` is a subclass of `evaluator` and implements the Jaro-Winkler distance metric developed (Winkler 1999) as an extension of previous work by (Jaro 1989; 1995) while working on ways to determine the probability that two family names were the same (with typos) in health records. The original Jaro distance metric computes string similarity $\Phi(s_1, s_2)$ based on the number m of characters appearing in the same order in both strings (of lengths $|s_1|$ and $|s_2|$ respectively), as well as the number of characters t shared by both strings, but requiring re-alignment. We therefore obtain the Jaro metric formula as represented in (Winkler 2006, §3.3):

$$\Phi(s_1, s_2) = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right)$$

This is normalised since $0 \leq m \leq |s_1|$ and $0 \leq m \leq |s_2|$ (because the maximum number of shared characters is the size of the shortest string), and $0 \leq m - t \leq m$ trivially since t is positive or 0, hence $\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) < 3$, thus $0 \leq \Phi(s_1, s_2) \leq 1$, and so the metric is normalised.

The Jaro-Winkler metric adds simply two factors: the length b of the shared prefix at the beginning of the string up to a maximum of 4 (essentially the

[4]Available at `http://code.michael-noll.com/?p=third-party;a=tree;f=python-Levenshtein-0.10.1`. Provided free under the GPL. Author unknown.

number amongst the first $n < 4$ characters shared by both strings) and an empirically determined scaling factor p usually set at 0.1. The Jaro-Winkler distance $\Phi'(s_1, s_2)$ thus becomes:

$$\Phi'(s_1, s_2) = \Phi(s_1, s_2) + b \cdot p \cdot (1 - \Phi(s_1, s_2))$$

which must be normalised whenever the scaling factor b is inferior to 0.25, since consequently $bp \leq 1$, hence $b \cdot p \cdot (1 - \Phi(s_1, s_2)) < 1$. Thus trivially $0 \leq \Phi'(s_1, s_2) \leq 1$ for $b \leq 0.25$.

The metric is implemented simply by overloading `scoreline`, extracting the document strings, passing them to the function `jaro` which returns the Jaro-Winkler distance, which is returned to the calling function. The function `jaro` is provided by the Levenshtein library described above.

5.2.4 Ratio Similarity

The class `jaroeval` is a subclass of `evaluator` and implements a superficial syntactic similarity called ratio similarity. If $l(s_1, s_2)$ is the Levenshtein edit distance of strings s_1 and s_2, the ratio similarity r_s is given by[5]:

$$r_s = \frac{(|s_1| + |s_2|) - l(s_1, s_2)}{(|s_1| + |s_2|)}$$

which is normalised since $0 \leq l(s_1, s_2) \leq 1$ and presumably the strings are non-empty.

This metric is implemented simply by overloading `scoreline`, extracting the document strings from the corpus entry, and passing them to `ratio` (provided by the Levenshtein Python library), which returns the ratio similarity, which is in turn returned to the calling function.

5.2.5 Jaccard Similarity

The class `jaccardeval` is a subclass of `evaluator` and implements the Jaccard index of set similarity, developed by (Jaccard 1901) while working on

[5]Due to lack of documentation for the Levenshtein Python library, if one wishes to verify this, one should check lines 721–734 of the `Levenshtein.c` file in the library.

the statistical distribution of types of flowers in the Alps. It measures the similarity of two sets by dividing the size of the intersection between two sets by the size of their union. This metric is normalised since the size of the intersection of two sets is at most the size of their union (namely when they are identical).

The implementation simply overloads `scoreline`, extracts the tokens from the corpus entry, removes all repeating elements within the token lists to form sets (using Python's `set()` built-in function) and passes the sets to `jaccard_distance`, a function provided by NLTK's metrics library, which computes the Jaccard Distance (the Jaccard similarity subtracted from 1) of the two sets. We subtract the result from 1 to recover the Jaccard index, and passes it back to the calling function.

5.2.6 MASI Distance

The final syntactic metric implemented is the MASI distance developed by (Passonneau 2006), and implemented in `masieval`, as subclass of `evaluator`. The MASI distance $masi(A, B)$ between two sets of labels A and B is simply 1.0 minus the length of the intersection of the sets, divided by the size of the largest set:

$$masi(A, B) = 1.0 - \frac{|A \cap B|}{\max(|A|, |B|)}$$

As we can see, it is a dissimilarity measure between sets ($masi(A, B) = 0$ if $A = B$, and tends towards 1 for dissimilar sets), which is normalised since size of the intersection of two sets is always lesser or equal to the size of the largest set, and set sizes are always non-negative, hence $0.0 \leq masi(A, B) \leq 1.0$.

The implementation overloads the `scoreline` function, extracts the document token lists from the corpus entry, forms sets out of them by removing repeated entries, and passes the document sets to the `masi_distance` function provided by NLTK, which computes the MASI distance between the two sets. We subtract this measure from 1.0 to obtain a similarity measure which is high for identical sets and low for different sets. The thus-obtained score is returned to the calling function.

5.3 BLEU: A Machine Translation Evaluation Metric

In this section, we discuss the background and implementation of the BLEU metric, from the world of machine translation. It exploits syntactic and lexical aspects of sentence pairs to judge whether one is a correct translation of the other. We will show, below, how this kind of evaluation was adapted as a similarity metric.

5.3.1 Origin and use

The BLEU (BiLingual Evaluation Understudy) metric was developed by (Papineni et al. 2001) as a method for evaluating the correctness of translation systems automatically, *i.e.* without the need for time-consuming human evaluation. For it to work, two components are required. The first is a set of reference translations of the sentences which the evaluated systems will be translating. For example, if we were evaluating how well a series of machine-translation systems would do when translating a corpus of French sentences, we would first have human translators produce a reference translation for each corpus entry. The second component is a numerical formula from which we can compute the correctness of a machine-translation of some sentence A based uniquely on the reference translation of sentence A. Because the metric only looks at the reference translation and the machine-translation, it is qualified as *language independent*, which suits our experiment well since we have parallel translations from French and from Chinese (*cf.* 'novels' and 'mtc' subsections of the paraphrase corpus section described in §4.5.2).

The 'naïve' baseline version of BLEU we are using here works as follows: we search for how many n-grams from a candidate translation appear in the reference translation and divide this by the number of n-grams in the candidate translation to obtain the precision p_n for the document pair, for the value of n. The BLEU-n score of a document pair (candidate translation vs. reference translation) is then computed by calculating:

$$\text{BLEU-}n = BP \cdot \exp\left(\sum_{i=0}^{n} w_i \log p_i\right)$$

where BP is a brevity penalty, which (Papineni et al. 2001, p315) suggest should be:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases}$$

where c and r are the lengths of the candidate and reference translations. The variable w_i in the BLEU equation is a weighting for each n-gram precision. (Papineni et al. 2001, p315) suggest a uniform $w_i = 1/n$.

5.3.2 Implementation and score normalisation

We have chosen to implement a simple version of BLEU-n—namely BLEU-1—in `BLEUeval`, a subclass of `evaluator`. The class overloads `scoreline`, extracts the document strings of the corpus entry passed to it, and writes them to two temporary files, which it formats by adding a newline character at the end of each file, and then a '`*`' character on the new line. This is done because the evaluation of the document pair is done by an external program `bleu` compiled from source code provided by Simon Zwarts of Macquarie University[6]. The program is called using the process-opening command `Popen` from Python's `subprocess` library, and given the names of the temporary files the documents were written to as arguments. The BLEU-1 score is then read from the command output using a regex, and cast to a float value, which is returned to the calling function as the similarity score (after the temporary files are removed).

5.4 Wordnet-based Metrics

We move onto the description of a class of DSMs exploiting lexical relations to determine document similarity, making use of the hierarchically-structured corpus *Wordnet*. Wordnet was first developed at Princeton by George Miller, and is described in more detail by (Fellbaum et al. 1998). The corpus structure is that of a tree, where each node is a concept (*e.g.* a cat), its children are hyponyms of the concept (*i.e.* X is a hyponym of Y if 'X is a Y' is true), and its parent is its hypernym (*i.e.* X is a hypernym of Y if 'Y is a

[6] This code is available at `http://web.science.mq.edu.au/~szwarts/Downloads.php` and is not provided under any specific license.

X' is true). Concepts may be nouns, verbs or adjectives, although we will not concern ourselves with adjectives here (because the metrics we use only support noun/verb matchings, as we will soon discuss). Words are said to have *synsets*, which are sets of concepts for which the word can stand (*i.e.* concepts the word can be synonymous to, hence '*syn*set'). Finally, it might be useful to note that concepts become more abstract and general as one progresses up the tree towards ancestor concepts, and more specific as one progresses down towards the leaves.

In this section, we will present the background and implementation of three metrics exploiting the structure of this corpus and its internal relations in order to compute lexical similarity.

5.4.1 Overview of Wordnet Metrics

The idea of using Wordnet as a basis for computing *word* similarity has been developed and discussed in several sources, many of which are mentioned by (Pedersen et al. 2004) within the context of presenting a Perl module for using such metrics. The three we will look at here are *path similarity*, *Wu-Palmer similarity* and *Lin similarity*. All three of these metrics are functions which, given two concepts (*not* documents) compute their similarity. We will show how to scale this to documents when discussing implementation.

Path similarity is the simplest of the three. A path between two nodes A and B in a tree graph is a list of nodes beginning with A and ending with B or vice versa, such that for each node in the list, it is connected to the next node in the list by one vertex (except for the last node, which as no 'next node'). The path similarity between some node A and B on a tree (such as the Wordnet tree) is the *inverse* of the length of *the shortest path* between A and B. It is normalised since dividing 1 by a positive integer always yields a result between 0 and 1. The idea is obviously that concepts 'close together' on the Wordnet tree are more likely to be lexically similar (path similarity score tends towards 1) than not. It should be noted that this metric, like the other two described below, can only compute the similarity between nouns, and between verbs, since noun and verb nodes do not share any common ancestry (and hence have no path from one to the other).

One problem with path similarity is that it does not into account the hierar-

chical relations between compared nodes, or where they are on the tree (as while this might be true for more specific concepts, a short path amongst abstract concept nodes high up on the tree might link two very different concepts). The two metrics that follow attempt to use the hierarchical nature of Wordnet for more precise similarity measurement.

The *depth $d(c)$ of a concept c* is taken to be the distance of the concept's node from the root node of the Wordnet tree. The Wu-Palmer similarity—conceived of by (Wu and Palmer 1994)—of two concepts A and B is determined as follows. First, their most immediate common ancestor node $lcs(A, B)$ (also called the *least common subsumer*) is found. Next the depth of that node is read, and finally this depth is scaled relative to the depths of A and B. The version of Wu-Palmer we will be using (from NLTK) will calculate it according to the following expression:

$$\text{Wu-Palmer Similarity}(A, B) = \frac{2 \cdot d(lcs(A, B))}{d(A) + d(B)}$$

which is normalised since the depth of the common ancestor to two nodes is necessarily less or equal to than the lowest of the children node depths, hence the fraction is lesser or equal to one (and trivially greater than 0).

Lin similarity—presented by (Lin 1998)—takes this approach one step further by examining the information content of the immediate common ancestor node $lcs(A, B)$ of two nodes A and B, and scaling it by that of A and B. The information content of a concept c is determined as follows. Using a training corpus, we look for all instances of a concept in the corpus. From this we determine the probability $p(c)$ of encountering the concept (simply its instance frequency over the size of the corpus). The information content $ic(c)$ of the concept is its negative log likelihood:

$$ic(c) = -\log p(c)$$

The version of Lin Similarity we will use scales the information content of $lcs(A, B)$ by the sum of $ic(A)$ and $ic(B)$ as follows:

$$\text{Lin Similarity}(A, B) = \frac{2 \cdot ic(lcs(A, B))}{ic(A) + ic(B)}$$

We will now describe how we used these metrics to form DSMs whilst presenting the implementation details.

5.4.2 Implementing Wordnet-based Metrics

We need not concern ourselves with the mathematics of the metrics described above, since NLTK's `wordnet` module will provide us with the same functions as described by (Pedersen et al. 2004) (and also provided in his `Wordnet::Similarity` Perl module). Our chief task is to get parts of speech of document tokens so as to be able to apply these metrics to them (since they only compare nouns to other nouns, and verbs to other verbs, and ignore the rest), and also conceive of a way of computing the similarity between two documents using metrics which will compute the similarity between words, as discussed in §3.2.3.

We began by writing a `TAGevaluator` subclass of `evaluator` which would extend `evaluator`'s initialisation by POS-tagging each document from each corpus entry and adding the token/tag pairs for each document in each corpus entry to `eval_data`, using a standard POS-tagger provided by NLTK (`nltk.pos_tag()`). It would also define a class list called `stopwords` populated with English stopwords read from NLTK's `nltk.corpus.stopwords` corpus. The rest of `TAGevaluator`'s initialisation function is as in `evaluator`, and calls `getscore` which calls `scoreline` in order to evaluate the corpus for a given metric.

We then wrote a general Wordnet evaluation class—`WNevaluator`, a subclass of `TAGevaluator`—which does all the work. It has two attributes, `metric` and `ic`, which are both initialised to '`None`' ('void type object' for Python). The idea is that this class would contain all the code necessary to run the evaluation when `metric` was defined as one of the three Wordnet metrics (and `ic` pointed to a text to use as information context for Lin Similarity), and that the individual metric classes would need to do nothing more but overload the value of `metric` (and of `ic`, if needed), and implicitly class `WNevaluator`'s `scorecorpus` function would be called by `getscore` during the initialisation function, since they would provided none of their own. If no metric was provided, `WNevaluator` will not initialise and will raise and exception.

Thus all that is done in the classes `pathsimeval`, `WUPeval` and `LINeval`—subclasses of `WNevaluator` standing for Path similarity, Wu-Palmer similarity, and Lin Similarity respectively—would be to overload `metric` as `path_similarity`, `wup_similarity` and `lin_similarity`, respectively (all

provided by NLTK's `wordnet` module). Additionally, `LINeval` also overloads `ic` by setting it to NLTK's `wordnet_ic` module object set to retrieve the British National Corpus already processed to serve as an information context (*i.e.* it is a dictionary mapping concepts to probabilities calculated from the British National Corpus).

The `scoreline` function common to all these metrics and defined in `WNevaluator` works as follows: first it collects all the nouns from both documents from the corpus entry passed to it into two lists (one per document), using the `getgroups` function which also filters out stopwords listed in class list `stopwords`. Then it gathers a list of all the concepts from each list and puts them into two dictionaries (one for each document) each mapping nouns to lists of concepts. We create a list of all possible pairs of nouns built picking one noun from each document. For each such pair we iterate through concept pairs using the noun-to-concept dictionaries and compute the concept similarity using the metric specified in `metric`. The best score for that noun pair is recorded in a dictionary mapping noun-pairs to best concept similarity scores.

The above steps are repeated for all the verbs in each document to obtain a dictionary mapping verb-pairs to the similarity measure of the best-matching concepts underlying the verbs of that pair. We write the scores from both dictionaries to a single list, take the top 10% highest scores and average them, and thus obtain the document similarity measure for the document pair. This score is returned to the calling function.

5.4.3 Fine-tuning: How Far is Too Far?

The method described above is fairly fine tuned. Our first scaling attempt was to collect all the concepts of all the nouns from both documents into two lists, take all the concept pairs constructed from taking one concept from each list, compute the similarity of each pair, and average over all the scores. But since there may be a dozen or so other senses for each noun other than that in which it was used, this brute-force method performs very poorly because most concept-pairs evaluated have nothing to do with the concepts expressed in the documents.

Another approach was to take a first sense heuristic, since when retrieving the concepts underlying a word using Wordnet, the most common senses

are usually the first ones in the list. We therefore could take the sense of a word to be the first one on the list, and proceed as above (now word pairs and concept pairs are effectively the same, since we have one concept per document word). This produced slightly better results, but ultimately they were still much worse than those provided by the 'top 10%' heuristic we described in §5.4.2.

The point being made is that more sophisticated scaling heuristics produce better results. We could doubtlessly do even better. For example, J. Carthy[7] suggests using *lexical chaining* with Wordnet to disambiguate a sentence (mapping words to precise concepts). If effective, this would most likely produce better results than our 'top 10%' heuristic since it essentially maps words to their concepts (with some error), and thus we compare concepts directly, as we would for words, and can safely average over the whole bunch.

Why, then, did we not do this? First of all, because it would have taken considerably more time to research how to do so correctly, and would involve straying from the priorities of this experiment by focussing on a small set of metrics. But more importantly, as we argued in §3.2.3, we wish to evaluate the original metrics, rather than our ability to derive new metrics from them. In order to compare the Wordnet-based metrics to other metrics, it was necessary to scale them so that they could process the same corpus. However, if the onus for results lay more in the heuristic used to disambiguate document words rather than evaluating their similarity, we would be performing a different task from that which was originally intended.

5.5 Semantic Vectors

In this section, we present the concept of distributional measures of semantic similarity, and show how we used a vector-base approach to distributional similarity measurement using a package designed by (Widdows and Ferraro 2008) to implement this metric.

[7] *cf.* Short bibliographical overview at `http://www.csi.ucd.ie/staff/jcarthy/home/Lex.html`.

5.5.1 Background Theory

Although the origin of distributional semantic similarity measures is usually attributed to (Firth 1957) (*cf.* §2.2.3 and §2.2.4) we argued in §2.3 that its origin was more likely the 'meaning-is-use' presented by (Wittgenstein 1953). The idea is to determine the semantics of a word by looking at *other words* used alongside it in English sentences.

One way of doing this is to use vectors to represent word senses, and to use *cosine similarity* of two vectors to compute the semantic similarity of the words they stand for. First, one must build a set of vectors for each word in our n-entries lexicon[8]. The vectors are set in an n-dimensional orthogonal vector space where each basis is assigned a unique lexicon word (hence each lexicon entry has a basis in the vector space). For each word in the lexicon, a vector is determined in this space, where the vector component for each basis is the number of instances the word the basis stands for appears in the context of the word a vector is being constructed for. 'Context' here can be as general as the word appearing in the same document, or as specific as some more fine-grained definition, such as that used by (Grefenstette 1992), who suggests using words with some grammatical relation to the target words to construct vectors (*e.g.* adjectives modifying the word, verbs it is the subject of, etc.). During this operation, we naturally ignore the cases of a word co-occuring with itself.

The similarity between two words is then determined by taking the cosine component of their vectors' product. If words A and B have vectors v_A and v_B, then the semantic similarity $sim(A, B)$ is simply:

$$sim(A, B) = \frac{v_A \cdot v_B}{|v_A| \cdot |v_B|}$$

which is normalised, since it is just a cosine reading of the unit vector component of v_A and v_B, which is a real number between 0 and 1.

To give a short example, let us suppose the only words in our keyword lexicon are 'cat', 'kitten' and 'dog'. We read that 'kitten' co-occurs with 'cat' in 12 documents, and with dog in only 3. We consider a vector space with only 'dog' and 'cat' as basis vectors, in which the vector kitten would be 3 times

[8] Usually stopwords and common verbs (if not already in the stopword list) are removed from the lexicon, since they do not aid sense disambiguation much, being frequently present

the 'dog' basis vector, and 12 times the 'cat' basis vector, thus its similarity with the 'cat' basis vector would be:

$$sim(\text{kitten}, \text{cat}) = \frac{3 \cdot 0 + 12 \cdot 1}{\sqrt{3^2 + 12^2} \cdot 1} = \frac{12}{\sqrt{153}} \approx 0.97$$

while its similarity with the 'dog' basis vector would be:

$$sim(\text{kitten}, \text{dog}) = \frac{3 \cdot 1 + 12 \cdot 0}{\sqrt{3^2 + 12^2} \cdot 1} = \frac{3}{\sqrt{153}} \approx 0.24$$

hence showing us that document co-occurrence has sufficed to show us that 'kitten' is closer to 'cat' than to 'dog' since it appears in the same context as the former more often than that of the latter.

5.5.2 Implementation: A Different Approach to Word-Sentence Scaling

The implementation of this metric was a bit more complicated than the previous ones. We used the `SemanticVectors` packaged developed by (Widdows and Ferraro 2008), which a Java library (the rest of our project is in Python). The `SemanticVectors` package provides a set of function used for building the vectors from a corpus, and helper functions for computing the scalar product of two vectors. Using these, and a script we provided to extract a plain-text version of the American National Corpus, Pascal Combescot, who was working on a similar project, provided us with a Java script which we adapted[9] into a command-line script taking two document files as arguments and returning their semantic similarity, as well as a vector set to use for evaluation based on the ANC. The source for this program has been reproduced in §B.5.

The reader may wonder how we integrated this into our evaluation classes, and more importantly: how did we compare documents in this manner when this vector-based distributional metric was designed to compare the meaning of *words*? To answer the second question, we simply took the vector of a document to be the sum of all the vectors standing for words contained in

[9] We re-wrote under half of the `main` function shown in §B.5 to adapt the class to our purposes. The rest of this Java package was written by Pascal Combescot.

the document (if words have no vectors, they are ignored). This gave good empirical results in Pascal's project, so we adapted this method here too. The obvious downside is that while fairly effective, it is fundamentally flawed: vector addition is commutative, hence for any words A and B, $v_A + v_B = v_B + v_A$. Therefore the vector for 'The dog bit the man' would be the same as that of 'The man bit the dog', effectively assigning them an identical sense even though the meaning obviously differs. However, this heuristic worked well enough, so we were curious to see how it would compare to other metrics and used it as such.

Finally, to integrate the measure into our evaluation system, we wrote `SemVecteval`, a subclass of `evaluator`. It overloads `getscore` (instead of the usual `scoreline` function). For each corpus entry, it writes the documents of the entry's document pair to two separate temporary files, the names of which it records in an index file along the ID of the entry. Once it has done this for each entry in the corpus, it passes the index file to the `SemanticVectorsEvaluator` Java command-line program based on Pascal's script described above, which computes the document similarity of each document pair in the corpus, writes it to a file mapping IDs to scores, which `getscore` reads once the process is completed in order to recover the metric scores, which it then stores in the usual `scores` dictionary mapping IDs to metric scores, which it returns to the initialisation function.

Chapter 6

Results and Analysis

In this chapter we will first briefly describe, in §6.1, how the analysis script was implemented according to the specifications discussed in §3.3. Following that, in §6.2 we will present the metric rankings and examine the CGS score breakdown—first discussed in §3.3.2—of a few metrics for the corpus categories where the scores were interesting or surprising.

6.1 Structure of Analysis Script

The analysis script—`analyser.py`—is a command-line utility which essentially takes a data file containing the results of evaluating all the metrics using the classes discussing in Chapter 5, and produces a file containing the metric rankings, as well as a series of other files. Its source code is presented in §B.7. The class takes as input a dictionary `results` mapping metrics to dictionaries which in turn map IDs to the metric CGS score for the corpus entries with those IDs. In short, to retrieve a DSM `metric` CGS score for corpus entry `ID`, one would call `results[metric][ID]`.

As discussed in §4.1.1, the ID for each corpus entry tells us which corpus section it is from, as well as what subsection (if any) and if it was generated by mismatching documents (mixed corpus entries) based a '(CORPUS_SECTION) [-SUBSECTION_TAG] (ID_NUMBER) [m]' structure. We use the information in the ID tags for all the corpus entries in results to pop-

ulate three dictionaries, `cats`, `matchcats` and `mixedcats` which will map category names to lists of IDs of corpus entries in those categories, with the help of regexes exploiting the structure of the tags. The difference between these dictionaries is:

- `cats` maps categories to lists of all the IDs of corpus entries in those categories.
- `matchcats` maps categories to lists of all the IDs of corpus entries in those categories which were generated by the corpus constructors (*i.e.* non-mixed).
- `mixedcats` does as above, but the IDs are those of corpus entries generate through mixing.

These three dictionaries are passed, with `results`, one by one, to the function `getcatscores` which assembles a dictionary mapping metrics to dictionaries mapping categories to dictionaries mapping tags to CGS scores. The dictionaries are stored in `catscores`, `matchcatscores` and `mixedcatscores`. They are essentially `results` with some extra classification: *e.g.* `matchcatscores[metric][ca` gives you a dictionary mapping all the IDs in the category `category` to CGS scores for DSM `metric`.

These three new dictionaries are then passed, one by one, to `getmetricAVGs` which returns three separate dictionaries—`metricAVGs`, `matchmetricAVGs` and `mixedmetricAVGs`—mapping categories to dictionaries mapping metrics to their average CGS score for that category. To clarify, if one wanted the average CGS score of DSM `metric` for documents of category `cat`, but only those produced by mixing, one would call `mixedmetricsAVGs[cat][metric]`. If one wanted the average CGS score for all documents in the category, one would call `metricsAVGs[cat][metric]`.

These dictionaries are first passed to the function `publishAVGs` which publishes the CGS averages per category per metric to a specified file (`results.txt` in our experiment). Next they are passed to the function `publishDists` which for each category, for each metric, publishes plots of the distribution of CGS scores for that metric, for documents of that category. We used Matplotlib's `pylab`[1] wrapper to achieve this.

The final step is to calculate metric differences—the MD score, as discussed

[1] Obtained from `http://matplotlib.sourceforge.net/`.

in §3.3.2. A list of all possible metric pairs is written to `metricpairs`. For each metric pair, the absolute difference between the metric CGS scores for each corpus document pair is written to the dictionary `metricdiffs` mapping metric pairs to dictionaries mapping corpus IDs to MD scores for that metric pair, for those corpus items. This dictionary is then used to publish the ranked MD scores to a text file (`metricdiffs.txt` in our experiment) using `publishDiffs`, a variant of `publishDists`. It also plots the distribution of MD scores for each metric pair, for each corpus section, using `publishDiffDists`, a variant of `publishDists`.

6.2 Discussion of Metric CGS Scores

The full rankings for average CGS scores per corpus section and per DSM are tabulated in the Tables 6.2–6.4. The totality of plots generated by the analysis script as well as the MD score rankings per corpus section and per metric pair are not provided here[2], as there are 2457 plots (338 for CGS score distributions, 2094 for metric comparisons) and a large number of MD rankings (namely the number of DSMs squared times the number of testbed corpus sections). In this section, we will discuss the CGS score rankings, and display charts where interesting results were found. We will leave discussion of how to exploit metric comparison scores for the conclusion.

6.2.1 Overall Rankings

As stated above, our experiment generates 338 plots[3], which is too many to examine each one individually. We will be relying on a the rankings to identify metrics which require closer investigation for certain corpus sections.

We have reproduced the CGS scores as percentages (*i.e.* we multiply the CGS score by 100 and round to the second decimal place) of closeness to a

[2]All files used for and generated by this experiment will be provided on an accompanying CD-ROM. See Appendix A for details.

[3]13 metrics times 9 document sections times 3 class of comparison (all corpus entries, only those generated by constructors, only those generated by mixing), minus the 13 entries for the mixed comparison class for the random edits corpus, which does not generate a mixed class (*cf.* §4.2.2).

Key	Corpus Category
1	Abstracts
2	Edits
3	POSswitch
4	Paraphrase
5	Paraphrase [MTC]
6	Paraphrase [News]
7	Paraphrase [Novel]
8	Themes
9	Wikipedia

Table 6.1: Corpus Category Legend

perfect evaluator in Tables 6.2, 6.3 and 6.4[4]. Hence a metric with a score of 89.02% for some corpus section is 82.02% as effective as a perfect DSM in identifying the kind of similarity that section exemplifies. For formatting reasons, the corpus section names in these tables were replaced by numbers. Refer to Table 6.1 to match numbers to corpus section names.

We will go through the rankings corpus section by corpus section. Notable scores will be highlighted in the score tables when referred to.

6.2.2 Random Edits Analysis

First up is the random edits section (*cf.* §4.2.2). We note that the Levenshtein edit distance scores well for all documents at over 90% CGS in Table 6.2. Looking at Figure 6.1 gives us a good example of what good metric results look like, with a majority of metric CGS scores for the individual section entries being very close to 1.0. This is unsurprising, since the edits used to generate the corpus are the exact same the Levenshtein edit distance tracks. Ratio similarity, as discussed in §5.2.4, makes use of the Levenshtein edit distance in addition to the size of the strings, so it is normal to find it in second place with a score of 82.64%. Jaro-Winkler distance also works well

[4] We note the absence of corpus section 2 (random edits) in this table. This is because all the corpus entries for this section were generated by the corpus builder, and no mixing took place. As such, the scores in the matched document scores table (*cf.* Table 6.3) are the same as in the overall scores table (*cf.* Table 6.2).

	1	2	3	4	5	6	7	8	9
BLEU	49.89	68.62	**80.44**	78.42	78.53	77.98	78.87	53.70	50.12
Character Count	49.88	65.32	70.73	59.30	60.22	52.98	66.59	**57.60**	49.18
Jaccard Distance	51.07	64.65	74.57	**75.25**	75.15	75.34	75.25	52.76	54.65
Jaro-Winkler	50.21	**74.61**	62.44	58.83	60.37	54.72	62.62	52.96	50.47
Levenshtein	49.94	**91.05**	75.18	70.35	69.94	68.87	72.67	51.74	51.32
MASI Distance	51.04	69.30	79.53	**77.26**	77.74	76.21	78.15	**54.26**	54.61
Random	50.53	67.81	48.61	50.95	51.56	50.15	51.36	50.84	52.39
Ratio	49.91	**82.64**	72.41	68.24	68.72	65.72	71.04	52.51	51.89
Semantic Vectors	**56.08**	66.29	73.83	**74.03**	72.92	72.97	76.50	53.03	**60.79**
WN Lin	50.71	70.60	71.80	66.86	65.23	64.42	71.66	51.37	52.03
WN Path	51.09	67.89	**80.40**	**77.73**	77.68	76.98	78.74	51.72	51.94
WN Wu-Palmer	50.41	**71.29**	62.44	52.20	47.27	47.94	62.65	50.77	51.07
Wordcount	49.98	53.36	54.54	45.94	46.69	43.33	48.58	51.24	49.92

Table 6.2: Overall Rankings (% CGS Scores)

	1	2	3	4	5	6	7	8	9
BLEU	0.40	68.62	**66.99**	64.85	65.77	64.10	64.92	17.71	0.84
Character Count	3.16	65.32	93.86	82.41	85.77	77.79	85.06	65.62	17.43
Jaccard Distance	5.47	64.65	54.53	59.22	58.18	58.92	60.66	14.51	15.04
Jaro-Winkler	49.89	74.61	79.09	79.07	83.34	73.34	82.24	68.11	54.78
Levenshtein	3.01	91.05	70.96	65.34	65.03	62.88	68.86	25.90	13.50
MASI Distance	5.58	69.30	67.25	67.86	67.76	65.79	70.64	21.72	16.39
Random	50.59	67.81	47.96	53.80	56.44	52.59	52.73	51.45	51.37
Ratio	5.76	82.64	77.27	75.00	76.61	71.12	78.43	39.92	21.55
Semantic Vectors	**83.53**	66.29	67.07	75.47	81.04	74.29	71.41	65.93	**88.42**
WN Lin	66.89	70.60	85.07	81.89	85.78	81.62	78.35	68.71	65.69
WN Path	34.42	67.89	**81.59**	78.54	80.52	79.40	75.44	36.71	34.54
WN Wu-Palmer	81.69	71.29	86.49	78.29	80.17	77.58	77.34	84.20	83.84
Wordcount	82.46	53.36	99.95	85.12	87.09	81.88	87.35	93.20	84.89

Table 6.3: Matched Document Pair Rankings (% CGS Scores)

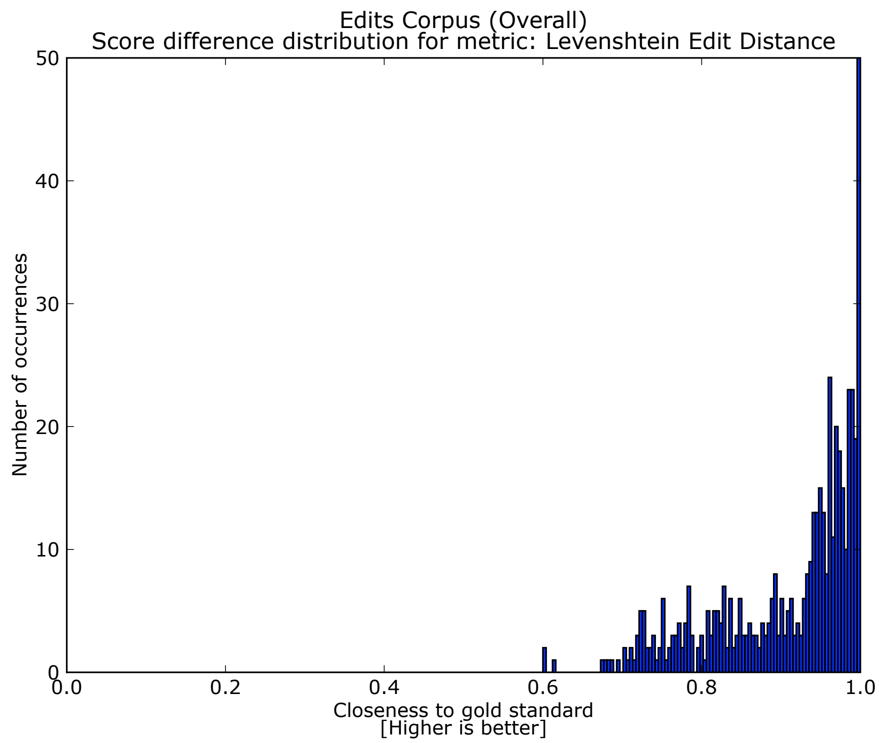


Figure 6.1: Levenshtein Edit Distance CGS Score Distribution (Edits Section)

enough because it makes use of shared characters between strings to determine similarity (and since one string is the edited version of the other, they generally share at least some characters), scoring 74.61%. The distribution of results (*cf.* Figure 6.2) starts to be a bit more spread out, indicating a less effective metric. One interesting point of note is that all the Wordnet-

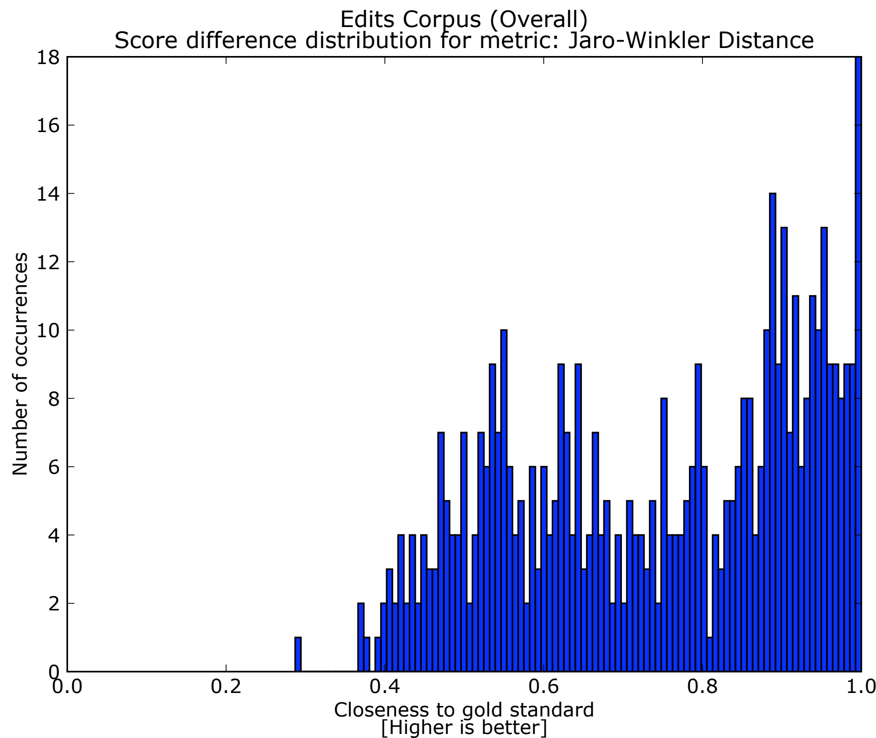


Figure 6.2: Jaro-Winkler Distance CGS Score Distribution (Edits Section)

based methods score better than random. We hypothesise that this may be because in many document pairs in the section, no edits are made to keywords the Wordnet metrics uses to determine similarity, and thus it matches a significant number of keywords from sentences that are not heavily edited (thus similar), thereby scoring these sentence pairs highly and obtaining a high CGS score for these; and that it ignores most of the words in heavily edited (thus dissimilar) pairs, thereby giving them a low similarity score (and obtaining a high CGS score again).

6.2.3 POS-Switch Analysis

The other syntactic toy-corpus section, POS-Switch (*cf.* §4.2.3), yields two particularly interesting results which exemplify proper analysis methodology. The top-two ranking metrics are BLEU and Wordnet Path similarity, both scoring highly in Table 6.2. We hypothesise that this is for roughly the same reasons as in the random edit corpus: dissimilar sentence pairs with many substitutions will score poorly on the BLEU metric (since one pair is the reference translation for the other, and randomly substituted words are bound to lower the BLEU score, as well as the Path Similarity since they are unlikely to be lexically related to the word they replace). At face value, they may look the same, but further analysis tells us otherwise: if we look at the entries for these metrics in the POS-switch section of Tables 6.3 and 6.4, we notice that while the score for Path Similarity in both these tables is roughly the same as the original, the score for BLEU is considerably lower for match documents, and higher for mixed (mismatched) documents, suggesting that BLEU picks up more strongly on some characteristic which affects document similarity but doesn't fully determine it. The score disparity is especially evident when one looks at the breakdown of score distributions for both metrics.

For the overall score distribution, seen in Figure 6.3, we note that although BLEU has a high CGS score density in the high-score bracket (above 0.8), it also has a has a more significant distribution of scores across the score range. Path Similarity, on the other hand, has a very sharp spike for the top score (meaning the CGS score for almost all section documents is 1.0). It, however, makes more drastic errors for a set of documents (notice the spike at the 0 CGS score point). When we examine how the metrics perform for the matched section entries in Figure 6.4, there is not much change to the CGS score distribution for Path Similarity, however we observe that the CGS scores for BLEU are spread widely across the centre of the score range, with several sharp spikes at CGS values inferior to 0.5. Finally when we examine how the metrics perform for the mixed section entries in Figure 6.5, we see a heavy concentration of high scores for BLEU, while the score distribution for Path Similarity retains the same shape. We judge from this that document features causing lowering of CGS scores for Path Similarity are *dissociated* from those used to assess similarity by Path Similarity, whereas BLEU correctly identifies features that make documents dissimilar, but is

not as good at identifying similar documents, and thus is exploiting features that don't solely determine similarity. The result of this in-depth analysis is that although both DSMs achieve similar performance in this testbed section at first blush, graphical analysis of the CGS score distributions allow us to assess that Path Similarity is a better metric for this corpus section than BLEU.

6.2.4 Theme Analysis

We now move on to our lexical similarity section—the Theme section (*cf.* §4.3). This corpus section produces no truly interesting results, given the metrics we evaluated. However, the top two metrics do have interesting score distributions which we will use to illustrate the danger of thinking that a metric with a higher average CGS score is necessarily a more sophisticated one. As can be seen in Table 6.2, character count scores 57.60% CGS, a few percentage points ahead of MASI distance, which scores 54.26%. However, consider the section CGS score distributions for these metrics, shown in Figure 6.6. We see that character count's CGS score distribution is close, in shape, to that of the Random Evaluator, also shown in Figure 6.6. This can serve as an indication that character count's high performance in this test is possibly a fluke and that it assigns metric scores fairly randomly. This intuition is confirmed by our knowledge of character count's *modus operandi* and the fact that although the chunks of paragraphs in the Theme section of the corpus are roughly the same length, there is much room for divergence in character count between the two documents of a corpus entry. The point here is that we could have deduced that the metric does not exploit any pragmatically significant features of documents relative to the type of similarity (namely lexical) we are concerned with, even if we had no knowledge of how the metric works or the structure of the corpus section. On the other hand, MASI forms two neat clusters in high-scoring and low-scoring regions of the plot, indicating it *at least* exploits some property common to all items of the corpus, but doesn't necessarily play a significant role in determining similarity. This thus gives us information about how metrics of this sort might function even without knowing its internal mechanisms, and also tells us that it might serve as a possible component for hybrid metrics, something Character Count obviously would not be suitably for, with regards to the type of similarity expressed in this corpus section.

6.2.5 Wikipedia Analysis

Next we examine the results of DSMs tested against the Wikipedia article corpus (*cf.* §4.4). This was a difficult corpus section, since articles were paired by subject and thus by loose lexical/semantic similarity, but the structural and lexical correspondence between the articles varies widely. As can be see in Table 6.2, no metric does particularly well. However, analysis of the CGS score distribution for the leading DMS—Semantic Vector Similarity (60.79%) can give us some indication as to how we could construct metrics capable of dealing with this corpus section (something the rankings alone don't tell us). Figure 6.7a shows two groupings of CGS scores: a high density grouping of entries with high (over 0.8) scores, and a spread of moderately low scores (in the 0.15 to 0.6 range). Contrary to the symmetric twin spikes observed for MASI, for the Theme section (*cf.* Figure 6.6b), the groupings here are largely asymmetric relative to one another. Also unlike the MASI case, a look at the score distributions for matched and mixed documents only (*cf.* Figures 6.7b and 6.7c, respectively) tells us that the Semantic Vectors metric is very good at detecting similarity for matched cases, making few mistakes (Figure 6.7b), but is also latching onto some similar features for the dissimilar cases, although not as strongly, since the distribution of errors in Figure 6.7b is broad and spreads into the better-than-random class of scores (over 0.5). This indicates that at least some of the properties semantic vector similarity is exploiting to judge similarity are present in the other documents. This, combined with our knowledge of how semantic vector similarity works could help us design a new metric for this sort of similarity. We know that the semantic vector metric adds up the word-vectors of all non-stopwords to build document vectors, which are then compared using cosine similarity. The problem, we can infer from the CGS breakdowns, is that there are some words being used to build document vectors which are in too many of the Wikipedia articles and are playing an unwanted role in the comparison of mismatched articles. We could fix this in several ways. One way we suggest would be to determine what non-stopwords are frequent throughout the Wikipedia corpus (what 'frequent' means here would have to be empirically determined, we suppose) and add these to the stopword list, since this 'lexical baseline' offers no significant contribution to similarity judgements. Naturally this sort of conclusion could be arrived at through thinking about how the metric works and how the corpus is structured in the

first place, but this example seeks to illustrate how graphical CGS analysis simplifies the task for us.

6.2.6 Paraphrase Analysis

We then look into the results for the Paraphrase section of the corpus (*cf.* §4.5) and its subsections. We read from Table 6.2 that the top scoring metrics are BLEU, Path Similarity, MASI and Jaccard Distances, and Semantic Vector Similarity, all scoring well above 70%. Levenshtein also does significantly better than random, scoring 70.35% overall. The is a lot to say here: let us begin by setting BLEU aside. Its high performance for paraphrase on the whole is no surprise, since it is *designed* to evaluate paraphrase generated through translation (which is what constitutes two subsections of the Paraphrase corpus section), and therefore it is not unexpected that it work well as an evaluator for general paraphrase as provided in the MTC subsection.

Let us instead focus on the other two syntactic metrics. Analysis of both of these metrics is straightforward. We look at the overall score distribution of both metrics in Figure 6.8 and notice that there is a high spike towards 1.0 CGS for both metrics, and that almost all the rest of the scores are in the high end of the 0.5–1.0 range, indicating that the metric probably scoring fairly consistently, *i.e.* that it predominantly exploits properties that determine similarity to produce its own judgement of similarity. All one needs is a quick glance at Figures 6.9 and 6.10 to confirm this, as we observe that the distribution of the CGS scores is the of the same shape as the overall distribution for both metrics, indicating that neither metric strongly relies on document properties that do not affect classification, since it makes no significant consistent error in either case (as is the case for metrics showing symmetrically opposed groupings in the overall distribution). In both cases, visual analysis quickly tells us that both DSMs are well designed for such a similarity type, and that better results are most likely to be obtained through improved heuristics in the metric design.

The 'surprises' in this corpus section are the performances of some of the syntactic measures. Let's examine the two top: MASI and Jaccard Distances. A quick glance through the CGS score distributions for overall, mixed and matched document pairs in Figures 6.11, 6.12 and 6.13 shows distributions that are fairly similar in shape and high-score density to the distributions

for semantic vector similarity and path similarity. Furthermore, we observe that the shapes of the distributions when considering only the matched and mixed documents are, in both instances, roughly the same as for the overall distribution (much like semantic vector similarity and path similarity's distributions were). We therefore can draw the same conclusions as for semantic vector similarity and path similarity: these metrics, which we previously thought to be purely syntactic, actually exploit document properties that are *similarity features*. Now, if we think back about how both these metrics work, they merely compute the intersection of sets of tokens (one for each document) and normalise it by the size of the union of the sets or by the size of the largest set for Jaccard Distance and MASI distance, respectively. We can hypothesise that when paraphrasing sentences, the structure does not change much, hence certain stop words are likely to occur no matter how we rephrase key words. Other words such as names are unlikely to change during paraphrase, and certain verbs and nouns may be present in both documents, hence MASI and Jaccard scores for paraphrase document pairs often come close the gold-standard score despite not being designed to track semantic or lexical properties.

That MASI and Jaccard distances could be used in such a way is fairly counter-intuitive, but the fact that they produce results on par with semantic metrics goes to show there is some evidence in support of the Wittgensteinian idea of abolishing distinctions between document similarity categories, as discussed in §2.3. Also, it illustrates how this analysis framework is powerful in both discovering new uses for metrics, and assessing their performance in more detail, guiding our explanations as to how they produce unexpectedly good results.

6.2.7 Abstract-Article Analysis

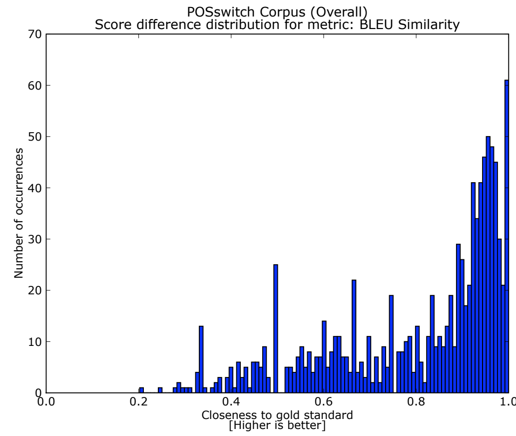
Finally we consider the abstract-article section of the corpus (*cf.* §4.6). For this testbed section, we will be brief, as none of the metrics have performed much better than the random evaluator. The only metric which stands out is Semantic Similarity.

As shown in Figure 6.14, the CGS score distribution for semantic vector similarity shows the characteristic symmetrically-opposed groupings that indicate that whatever properties the metric is exploiting aren't fully connected

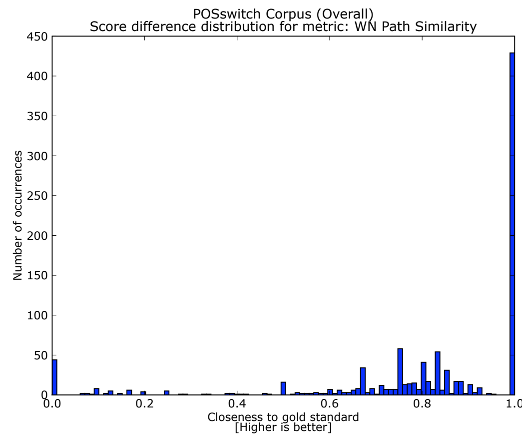
to similarity features, and are a cause of consistent misclassification. A quick glance at the score disparity between mixed document score (83.53%) and matched document score (28.64%) in Tables 6.3 and 6.4, respectively, provides us with information to determine the cause. The problem is similar to that which Semantic Vector similarity faced in the Wikipedia corpus: there is a 'baseline' vocabulary shared by most papers from the same subject or domain, which will be contribute to the document vectors of most documents and abstracts. As a result, Semantic Vector Similarity is very good at detecting similar documents, but concludes that even dissimilar documents are similar because they possess baseline vocabulary features. With the knowledge, DSM designers can consider the same approach we suggested for the Wikipedia corpus, and design metrics which identify shared vocabulary and ignore it (or give it less impact during assessment) before assessing similarity.

	1	3	4	5	6	7	8	9
BLEU	99.39	**93.88**	91.99	91.30	91.86	92.83	89.68	99.40
Character Count	96.59	47.60	36.19	34.66	28.17	48.13	49.58	80.93
Jaccard Distance	96.67	94.61	91.28	92.12	91.75	89.83	91.02	94.27
Jaro-Winkler	50.52	45.80	38.59	37.40	36.11	43.00	37.82	46.16
Levenshtein	96.88	79.40	75.35	74.86	74.86	76.47	77.58	89.15
MASI Distance	96.51	91.81	86.66	87.72	86.63	85.65	86.80	92.82
Random	50.48	49.27	48.09	46.68	47.72	49.99	50.24	53.42
Ratio	94.06	67.56	61.48	60.82	60.31	63.65	65.10	82.24
Semantic Vectors	**28.64**	80.59	72.59	64.80	71.66	81.58	40.13	**33.16**
WN Lin	34.54	58.53	51.83	44.67	47.22	64.97	34.02	38.36
WN Path	67.75	**79.21**	76.91	74.84	74.56	82.04	66.74	69.33
WN Wu-Palmer	19.13	38.39	26.10	14.37	18.31	47.96	17.34	18.29
Wordcount	17.50	9.14	6.76	6.29	4.78	9.81	9.29	14.96

Table 6.4: Mixed Document Pair Rankings (% CGS Scores)

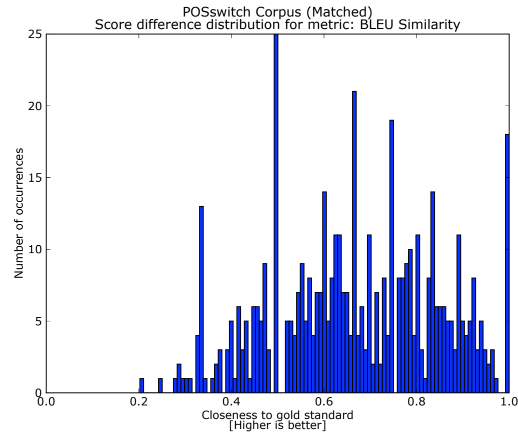


(a) BLEU CGS Distribution

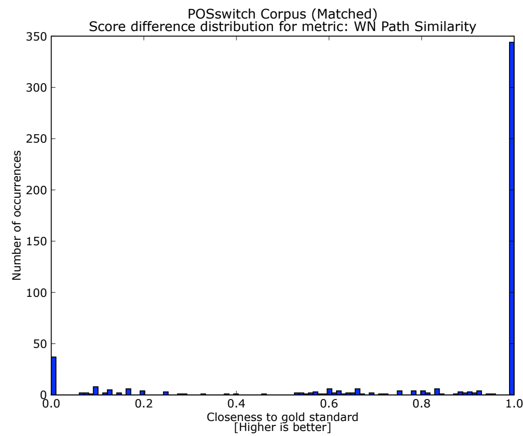


(b) Path Similarity CGS Distribution

Figure 6.3: Comparison of BLEU and Path Similarity for POS-switch Documents

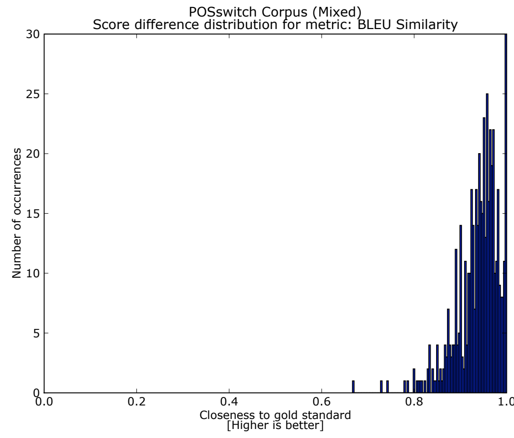


(a) BLEU CGS Distribution

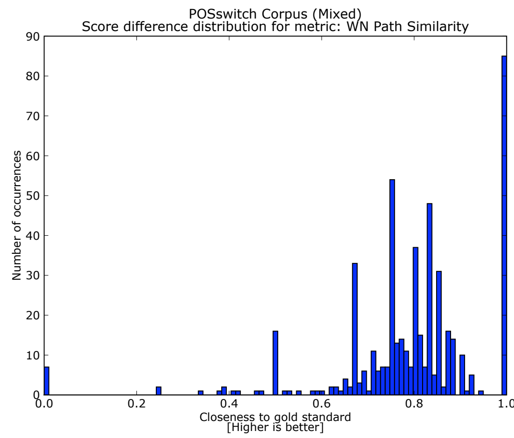


(b) Path Similarity CGS Distribution

Figure 6.4: Comparison of BLEU and Path Similarity for Matched POS-switch Documents

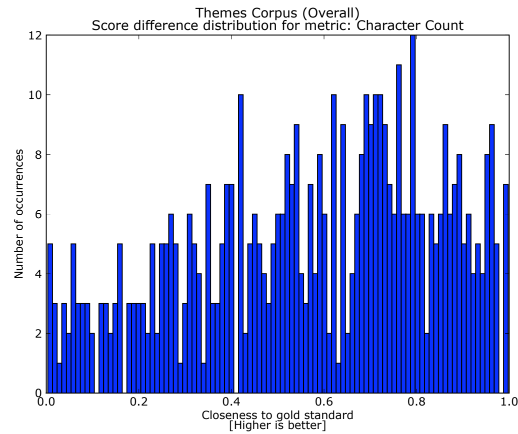


(a) BLEU CGS Distribution

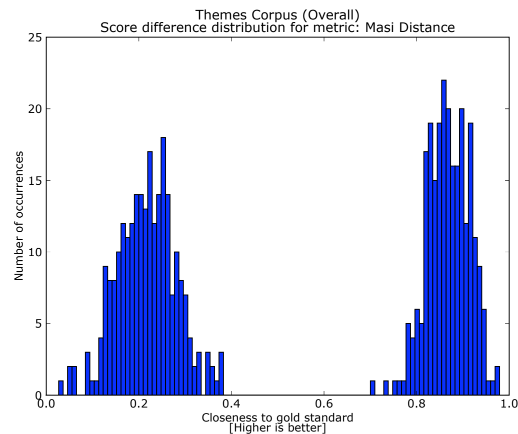


(b) Path Similarity CGS Distribution

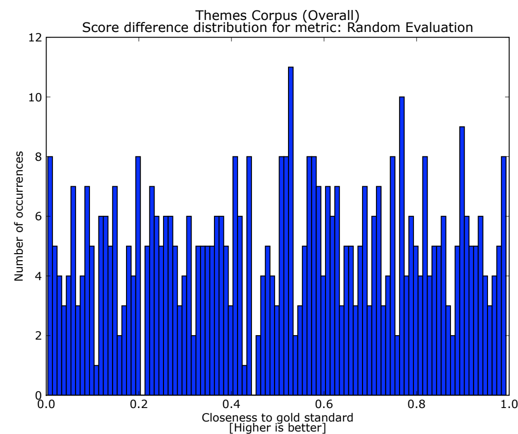
Figure 6.5: Comparison of BLEU and Path Similarity for Mixed POS-switch Documents



(a) Character Count CGS Distribution

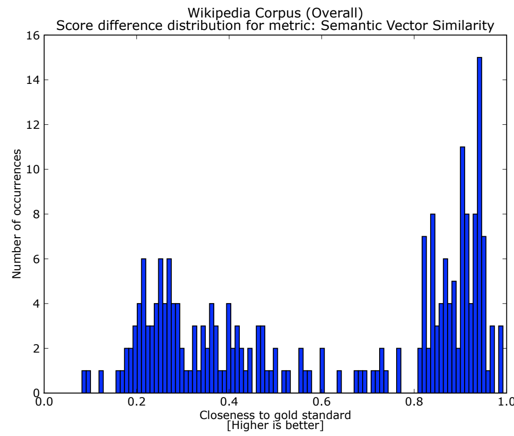


(b) MASI CGS Distribution

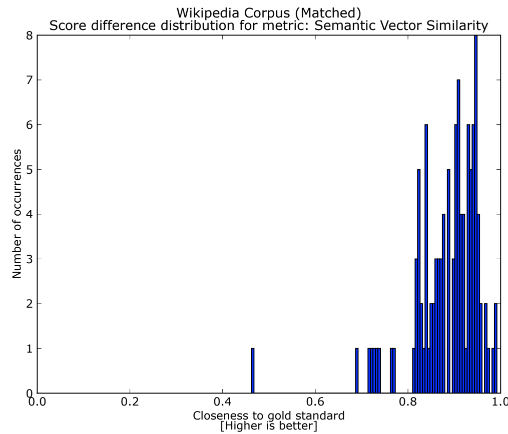


(c) Random Evaluator CGS Distribution

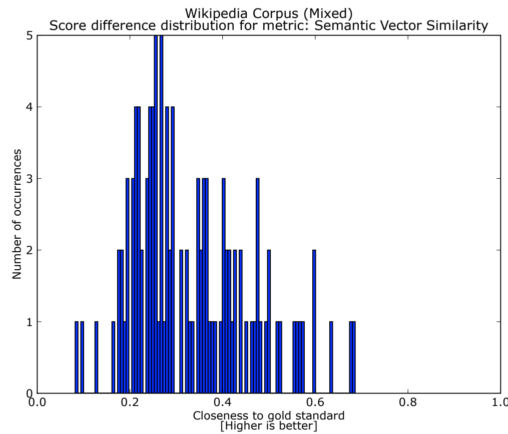
Figure 6.6: Comparison of Character Count and MASI for Theme Documents



(a) Semantic Vectors CGS Distribution (Overall)

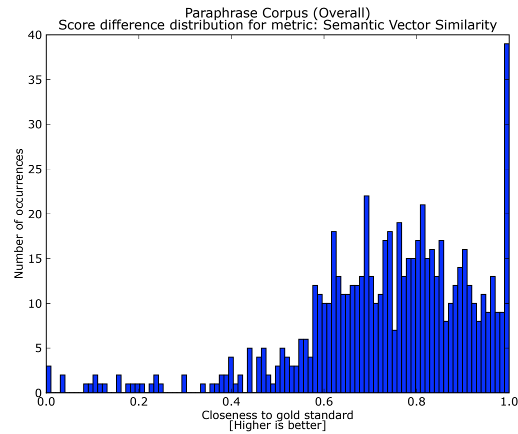


(b) Semantic Vectors CGS Distribution (Matched)

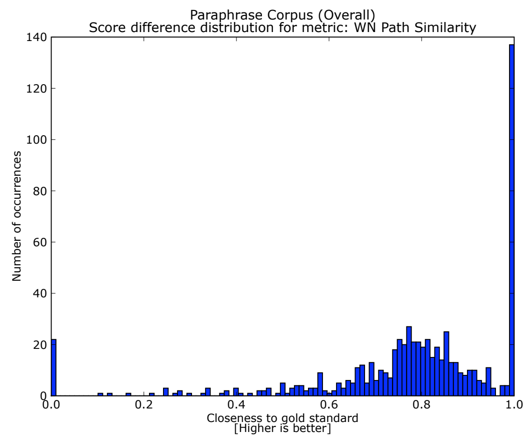


(c) Semantic Vectors CGS Distribution (Mixed)

Figure 6.7: Breakdown of Semantic Vectors CGS Score Distribution for Wikipedia Corpus Section

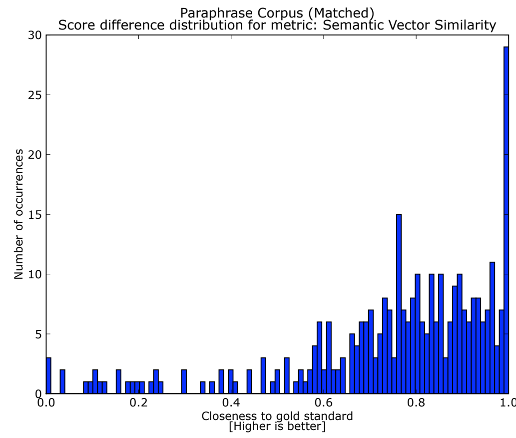


(a) Semantic Vectors CGS Distribution

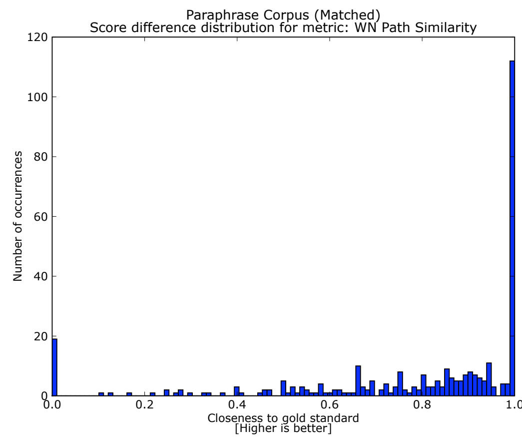


(b) Path Similarity CGS Distribution

Figure 6.8: Comparison of Semantic Vectors and Path Similarity for Paraphrase Documents

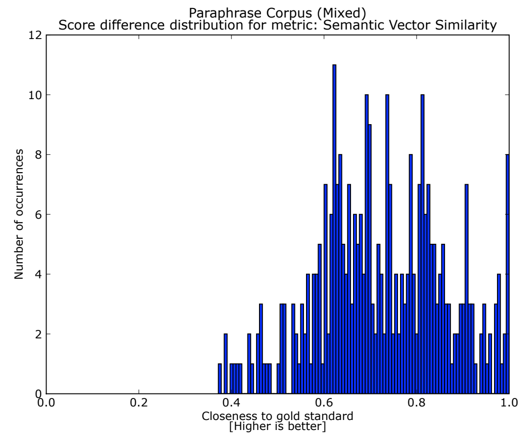


(a) Semantic Vectors CGS Distribution

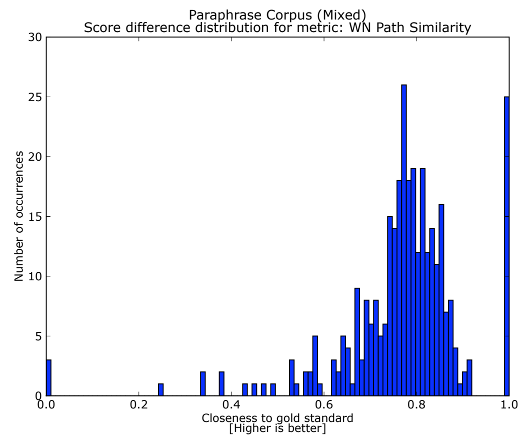


(b) Path Similarity CGS Distribution

Figure 6.9: Comparison of Semantic Vectors and Path Similarity for Matched Paraphrase Documents

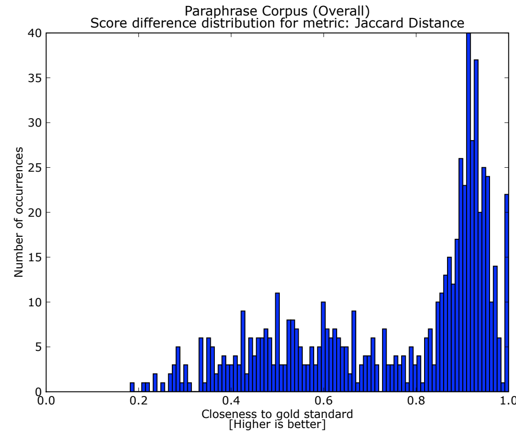


(a) Semantic Vectors CGS Distribution

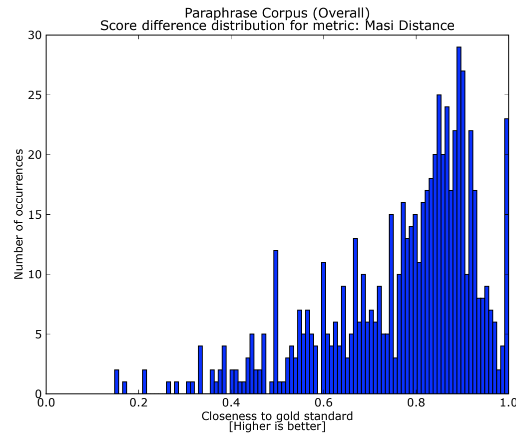


(b) Path Similarity CGS Distribution

Figure 6.10: Comparison of Semantic Vectors and Path Similarity for Mixed Paraphrase Documents

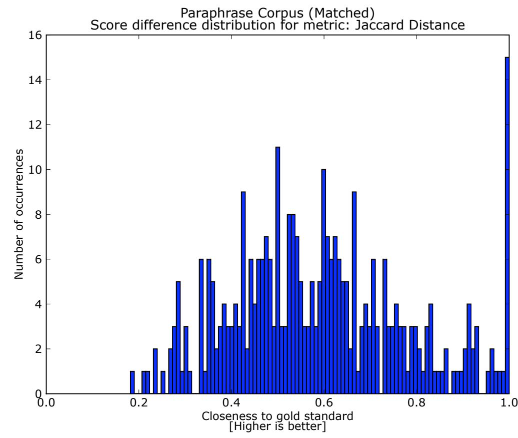


(a) Jaccard Distance CGS Distribution

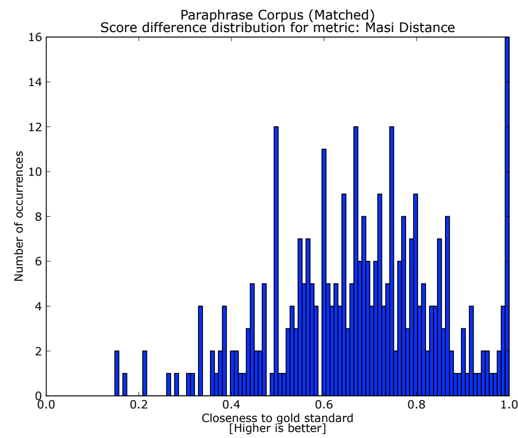


(b) MASI Distance CGS Distribution

Figure 6.11: Comparison of Jaccard Distance and MASI Distance for Paraphrase Documents

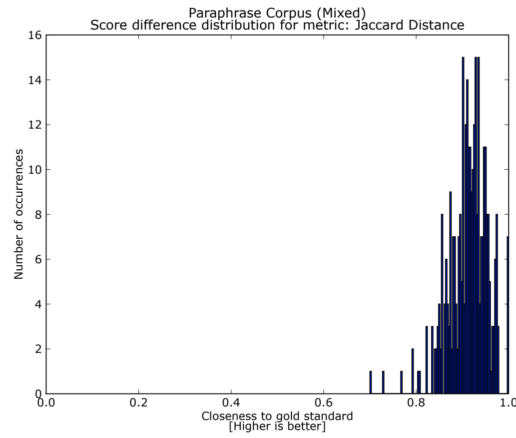


(a) Jaccard Distance CGS Distribution

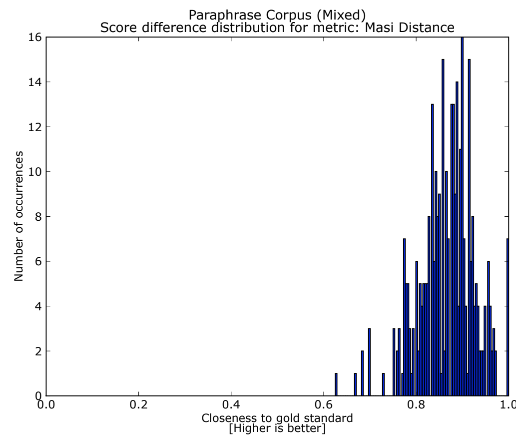


(b) MASI Distance CGS Distribution

Figure 6.12: Comparison of Jaccard Distance and MASI Distance for Matched Paraphrase Documents



(a) Jaccard Distance CGS Distribution



(b) MASI Distance CGS Distribution

Figure 6.13: Comparison of Jaccard Distance and MASI Distance for Mixed Paraphrase Documents

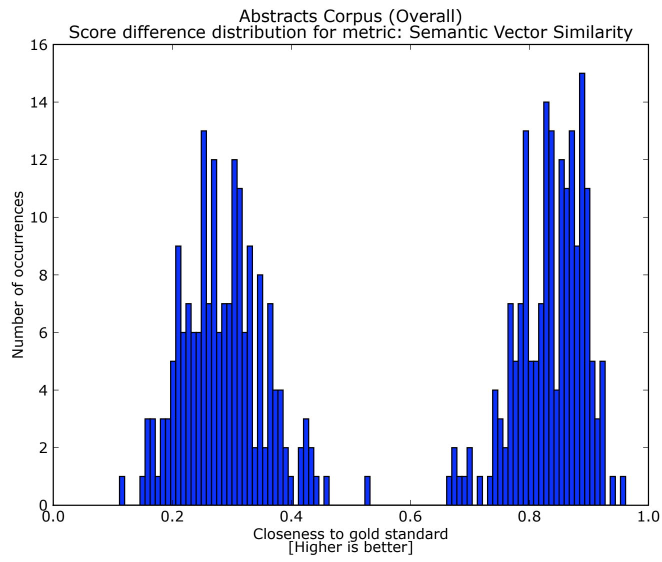


Figure 6.14: Semantic Vectors CGS Score Distribution (Abstracts)

Chapter 7

Conclusions

In the introduction to this work, we set out to answer three questions, thus setting three goals for this work. The first was to understand how common DSMs were designed and implemented, the second was to find a method for analysing and comparing DSMs, and the third was to determine how to improve DSMs based on the results of this analysis.

In this work, we began by giving a discussion of the theory underlying document similarity in Chapter 2, and in Chapter 3, we outlined the methodology for an experiment that would achieve these goals. In Chapter 4, we described how to generate a testbed corpus which would be used as a basis for further analysis as required by our second goal, as well as serve as an extensible framework for future experiments of this sort. In Chapter 5, we implemented a host of DSMs to be analysed in our experiment, additionally we integrated them into an easily extensible framework which could serve as the basis for further work by adding DSMs to be evaluated. Not only did this implementation work teach us a lot about using object-oriented design to create re-usable and extensible code, but the *conceptual* difficulties in implementing some of the metrics which needed scaling from word-based evaluation to document-based evaluation also provided a keen insight into the world of metric design, especially when considering how to improve the scaling heuristics for the Wordnet-based metrics in §5.4. In Chapter 6, we described the implementation of our analysis script, completing the description of the three-step analysis framework we had described in Chapter 3.

We then moved—in §6.2—onto the presentation and discussion of the results of our experiment, run using the frameworks described and designed in the previous chapters. The results for the different testbed sections, representing different types of similarity, revealed a variety of aspects of both DSMs and of the strength and benefits of the analysis framework that we had set out to show. First, we obtain validation for the Wittgensteinian ideas—presented in §2.3—of there being no fundamental difference between classes of similarity present in documents and evaluated by DSMs, as was shown when some lexico-semantic metrics were able to competently assess syntactic similarity in §6.2.3; but the reverse result was even more striking when, in §6.2.6, fairly simple syntactic metrics were able to perform not only competently, but on par with the best-performing semantic metrics we had. Thanks to the graphical aspect of our analysis, provided by plots of the CGS score distributions, we even able to give experimentally-motivated suggestions as to why these metrics performed well outside of their usual categories. These results, alongside the observations that many metrics worked well in the categories we expected them to, and that none of our—fairly elementary—DSMs performed exceptionally well for sections exemplifying complex notions of lexical and semantic similarity, as shown in §6.2.5 and §6.2.7, all go to demonstrate how our analysis framework provides an informative way of describing, comparing, and *explaining* the performance of DSMs. In short, we have—in the analysis chapter—come a long way towards achieving the second goal.

However, comparing metrics was not the only part of the second goal. We wanted to produce an effective method for analysing metric performance in a way that would help improve it. In §6.2.5 and §6.2.7, we used the graphical representation of the best-performing metrics to attempt to deduce the source of error that was impeding better DSM performance for the Wikipedia and Abstract-Article corpus sections, and were able, on the basis of this analysis, to make suggestions as to how to improve metrics in the face of such categories with complex types of similarity. In §6.2.4, we illustrated how the analysis framework allowed us to gain better understanding of how metrics worked even if we did not know the implementation details or corpus structure. Following that, in §6.2.6 we not only showed how visual analysis served us well in identifying the quality of metric performance better than the rankings alone (indeed our discussion in §6.2.3 shows how visual analysis of the CGS score distributions serves to demonstrate that better ranking metrics are not

necessarily the best, when considering the potential for improvement), but also we capable of giving suggestions for improvement, which works towards meeting our third and final goal: to determine how to improve DSMs based on the results of analysis.

We wish now to discuss how further work in this vein might proceed. We have, throughout this work, repeatedly mentioned the creation of hybrid metrics as an avenue for improving metrics based on the results of analysis. In §6.1, we discussed how in our implementation of the analysis script we included functions that would generate metric comparison rankings, where a metric pair would score higher on a 0 to 1 scale if the metrics in the pair produced largely different results for corpus entries, and lower is most metric results were in agreement. The complete rankings for metric comparisons are provided alongside other experimental data and source code on an accompanying CD-ROM (*cf.* Appendix A for details). We propose no specific methodology as to how to exploit these rankings, but the idea would to first search through our individual metric rankings for metrics performing well for some particular category, and if two or more such metrics exist, we refer to the metric comparison rankings. If amongst these high-performance metrics, there are two or more that have a high metric difference (MD) score (meaning that one correctly evaluates some of the errors of the other, and vice-versa), then we can use this information to attempt to produce better metrics.

The hybrid approach would be to combine two or more metrics judged to be different enough into a common evaluator, and use that as a metric. One way to do this would be to simply weigh metric scores by their individual performance and average over the scores for both metrics to obtain a combined metric result. Another would be to use machine learning methods such as genetic algorithms to determine how and when to combine metric scores, supplant one metric's judgements with those of the other, and so on. We leave it to researchers interested in further work to determine how to do this.

Another way of exploiting the results of analysis using metric differences would be to look at the documents where metric differ and determine why one metric gets better results than another. For example, if metric A get 80% CGS and metric B gets 70% CGS, but they differ on 20% of corpus entries for which we observe metric A has good results but metric B does not. We can then examine a sample of this 20% of the corpus for which metric A works but metric B does not. By comparing the documents with

those metric B works well with, and by considering the factors in the design of metric A which cause it to correctly assess this 20%, we may be able to deduce what design changes must be made to metric B in order to achieve the same score as metric A or conceivably up to 90% CGS. This is not, per se, a hybrid metric approach to DSM design, but it illustrates how there are many ways in which the analysis framework can be extended to help understand DSMs even better and assist in the creation of higher performance DSMs. We are confident that other researchers will be able to think of other ways of doing so using this framework as a basis to obtain even better results than the those produced by the metrics we used in this experiment.

In conclusion, not only did this project allow us to become more intimately acquainted with the issues, problems and aspects of designing document similarity measures, with their diversity of application and the rich theoretical background underlying their construction, but we believe we have achieved our additional goals of producing a framework suitable for analysing, comparing and ultimately improving document similarity measures that will be of use for future work, both ours and that of other researchers interested in the construction of better systems of comparison to use in a variety of text and language processing tasks.

Bibliography

Agirre, E., Alfonseca, E., Hall, K., Kravalova, J., Pasca, M. and Soroa, A.: 2009, A study on similarity and relatedness using distributional and wordnet-based approaches, *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, Association for Computational Linguistics, Boulder, Colorado, pp. 19–27.
URL: *http://www.aclweb.org/anthology/N/N09/N09-1003*

Allen, J.: 1987, *Natural Language Understanding*, The Benjamin/Cummings Publishing Company, Inc.

Bird, S., Klein, E. and Loper, E.: 2009, *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*, O'Reilly.

Bird, S. and Loper, E.: 2002, Nltk: The natural language toolkit, *Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*, pp. 62–69.

Blackburn, S.: 1996, *The Oxford dictionary of philosophy*, Oxford University Press Oxford.

Cantor, G.: 1874, Uber eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen, *Journal fur die reine und angewandte Mathematik* **77**, 258–262.

Cohn, T., Callison-Burch, C. and Lapata, M.: 2008, Constructing corpora for development and evaluation of paraphrase systems, *Computational Linguistics* **34**(4), 597–614.

Davidson, D.: 1978, What metaphors mean, *Critical Inquiry* pp. 31–47.

Dolan, B., Quirk, C. and Brockett, C.: 2004, Unsupervised construction of

large paraphrase corpora: Exploiting massively parallel news sources, *Proceedings of the 20th international conference on Computational Linguistics*, Association for Computational Linguistics Morristown, NJ, USA.

Fellbaum, C. et al.: 1998, *WordNet: An electronic lexical database*, MIT press Cambridge, MA.

Firth, J.: 1957, A synopsis of linguistic theory 1930-1955, *Studies in Linguistic Analysis* pp. 1–32.

Francis, W. and Kucera, H.: 1979, Brown corpus manual, *Brown University* .

Frege, G.: 1948, Sense and Reference, *The Philosophical Review* **57**(3), 209–230.

Frege, G. and Austin, J. L.: 1950, *The Foundations of Arithmetic (1884)*, 2nd edn, Blackwell.

Frege, G. and Furth, M.: 1964, *The Basic Laws of Arithmetic (1893)*, University of California Press.

Gärdenfors, P.: 2004, *Conceptual spaces: The geometry of thought*, MIT Press.

Grefenstette, G.: 1992, Use of syntactic context to produce term association lists for text retrieval, *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM New York, NY, USA, pp. 89–97.

Grice, H. P.: 1975, Logic and conversation, *Syntax and semantics*, Vol. 3, P. Cole and J. Morgan, pp. 41–58.

Griffith, T. and Ferrari, G.: 2000, *The Republic*, Cambridge University Press.

Jaccard, P.: 1901, Etude comparative de la distribution florale dans une portion des Alpes et des Jura, *Bull. Soc. Vaudoise Sci. Nat* **37**, 547–579.

Jaro, M.: 1989, Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida, *Journal of the American Statistical Association* pp. 414–420.

Jaro, M.: 1995, Probabilistic linkage of large public health data files, *Statistics in medicine* **14**(5-7), 491–498.

Levenshtein, V.: 1966, Binary codes capable of correcting deletions, insertions and reversals, *Soviet Physics Doklady*, Vol. 10, pp. 707–710.

Lin, D.: 1998, An information-theoretic definition of similarity, *Proceedings of the 15th International Conference on Machine Learning*, pp. 296–304.

Papineni, K., Roukos, S., Ward, T. and Zhu, W.: 2001, BLEU: a Method for Automatic Evaluation of Machine Translation, *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* pp. 311–318.

Passonneau, R.: 2006, Measuring agreement on set-valued items (MASI) for semantic and pragmatic annotation, *Proceedings of the International Conference on Language Resources and Evaluation (LREC).*

Pedersen, T., Patwardhan, S. and Michelizzi, J.: 2004, Wordnet:: similarity-measuring the relatedness of concepts, *Proceedings of the National Conference on Artificial Intelligence*, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, pp. 1024–1025.

Widdows, D. and Ferraro, K.: 2008, Semantic vectors: A scalable open source package and online technology management application, *Sixth International Conference on Language Resources and Evaluation (LREC 2008).*

Winkler, W.: 1999, The state of record linkage and current research problems, *Statistical Research Division, US Bureau of the Census, Wachington, DC* .

Winkler, W.: 2006, Overview of record linkage and current research directions, *US Bureau of the Census Research Report* .

Wittgenstein, L.: 1953, *Philosophical Investigations*, Blackwell Publishing.

Wu, Z. and Palmer, M.: 1994, Verbs semantics and lexical selection, *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, Association for Computational Linguistics Morristown, NJ, USA, pp. 133–138.

Appendix A

Technical Details

A.1 Supplied CD-ROM Contents

The running of the experiment described throughout this dissertation generated a lot of data: 237.3 megabytes—too much to include in an appendix. Also, while the essential source code for the extensible DSM evaluation framework we developed is available in Appendix B, it would naturally be helpful to provide all the code we wrote for this project, as well as the entirety of the Semantic Vectors package alongside the files used to evaluate similarity using distributional methods. What follows is a breakdown of the files we have provided on an accompanying CD-ROM[1]. The 'project directory' is taken to be the root folder containing all the files described below. For scripts to function correctly, the environment variable `$PROJECTLOC` must be set to the absolute path of this project directory on the user's system.

- `README.txt` — text file containing essentially the same information as found in the present appendix.
- `AnalysisDSM.pdf` — a PDF file of this dissertation.
- Folder `analysis` — contains all the experiment's output.

[1]In the eventuality that this CD-ROM is no longer accessible, the authors should be contacted (`edward.grefenstette@comlab.ox.ac.uk` or `egrefen@gmail.com`). We will strive to publish these files online too, and urge the interested reader to look us up on their favourite search engine.

  - Folder `Diffdists` — contains the plots for the CGS score distributions for each metric, organised into sub-folders by corpus section.
  - Folder `GSdists` — contains the plots for the MD score distribution for each metric pair, organised into sub-folders by corpus section.
  - `metricdiffs.txt` — rankings of average MD scores for all metric pairs, by corpus section.
  - `rankings.txt` — rankings of average CGS scores for all metrics, by corpus section.
  - `results.pkl` — pickled Python dictionary mapping metrics to dictionaries mapping IDs to CGS scores. Extract with Python `pickle` module.

- Folder `code` — contains all Python source code for the project.
  - `analyser.py` — the analysis script. Gives full usage instructions when called with option `-h`.
  - `buildCML.py` — a command-line interface for operating the corpus building framework. Gives full usage instructions when called with option `-h`.
  - `buildcorpus.py` — a script to run the corpus building phase of the experiment. All information is hard-coded relative to the project directory path. No arguments needed.
  - `corpusbuilder.py` — the corpus evaluation framework component.
  - `corpusevaluation.py` — the metric evaluation framework component.
  - `evalCML.py` — a command-line interface for operating the metric evaluation framework component. Gives full usage instructions when called with option `-h`.
  - `evaluatecorpus.py` — a script to run the metric evaluation phase of the experiment, using the metric evaluation framework. All information is hard-coded relative to the project directory path. No arguments needed.

  - `prepcorpus.py` — a script to pre-prepare a corpus for evaluation. Gives full usage instructions when called with option `-h`.

- Folder `corpus` — contains various versions of the corpus used for evaluation.

  - `docsimcorp.pkl` — a pickled object containing the entire corpus in Python dictionary format. Extract with Python `pickle` module.

  - `preppedcorp.pkl` — a pickled object containing the entire *prepped* corpus in Python dictionary format. Extract with Python `pickle` module.

  - `textcorpus.txt` — a plain text file containing the entire corpus. Each entry is spread over three lines, delimited by XML-like tags indicating which line is the first document (`doc1`), the second document (`doc2`) or the CGS score (`score`), as well as the entry ID.

- Folder `data` — contains source data used for the construction of various corpus sections.

  - Folder `LapataCorp` — contains the files used for the generation of the paraphrase corpus section (and subsections).

  - Folder `papers` — contains the PDFs used for the generation of the abstract-article corpus section.

  - `wiki_articles.txt` — list of English Wikipedia URLs used for the construction of the Wikipedia corpus section.

- Folder `scripts` — scripts used to run the experiment, to avoid the need for any interaction with the source code. The script files must be made executable by the user before being used (`chmod +x` *`filename`* on a UNIX system).

  - `buildcorpus` — builds the experimental corpus.

  - `evaluatemetrics` — evaluates metrics against experimental corpus.

  - `prepcorpus` — pre-processes the experimental corpus to prepare for evaluation.

  - `runall` — runs all other scripts, and hence the whole experiment.
  - `statanalysis` — runs complete analysis script.

- Folder `Semantic Vectors` — set of Java classes to run Dominic Widdows' Semantic Vectors package as a metric. Important files include:
  - Folder `lib` — library elements required to run classes. These files should be added to the user's Java class path.
  - `SemanticVectorsEvaluator.java` — main semantic vector evaluation class, originally written by Pascal Combescot and modified to fit the purpose of this project.
  - `termvectors.bin` — compiled semantic vectors for English words, constructed by Pascal Combescot using the American National Corpus.
- Folder `tools` — other tools used for this project, namely only `Bleu.cpp`, Simon Zwarts' implementation of BLEU. Needs to be compiled for the user's system.

A.2 System Requirements

For the framework to run, the user will require Python 2.5 or higher, as well as the `pylab`[2] package and dependent libraries, `nltk`, the `Levenshtein` package, the `Bleu` package, Widdows' `Semantic Vectors` package, which may need an up-to-date version of Java and may have other dependencies, and finally the Poppler PDF library. The sources for all these packages are provided when they are first mentioned in the dissertation. With these installed, all project components should work fine on UNIX-like systems, or in Windows by using cygwin. The authors will do their best to help with troubleshooting if needed[3].

[2] Available at `http://matplotlib.sourceforge.net/`.
[3] Please use the contact email addresses provided in §A.1 only.

Appendix B

Source Code

B.1 corpusbuilder.py

```
"""A collection of corpus generating classes.

Written by Edward Grefenstette, University of Oxford, 2009.
email: <edward.grefenstette@balliol.ox.ac.uk>
Submitted towards completion of MSc Computer Science.
"""

from __future__ import division
import re
from nltk import clean_html
import urllib2
import glob
import sys, os
from subprocess import call
import random
from nltk.corpus import brown

try:
    import psyco
    psyco.full()
except ImportError:
    pass

globalindex = 1

class corpusbuilder(object):
    noMix = False
    __tag=""

    def __init__(self):
        self.documents1 = {}
```

```
        self.documents2 = {}
        self.IDs = []
        self.corpus = {}
        self.mixedcorpus = {}
        self.mixedIDs = []

    def calcsimscore(self,index):
        """Get simscore for document pair at index.

        Some heuristic could be inserted here.
        Default is to assume complete similarity.
        Overload this function to add granularity.
        """
        return 1.0

    def scoreCorpus(self):
        """Get pairs of annotated sentences with their similarity score.
        """
        annotTuples = {}
        for ID in self.IDs:
            annotTuples[ID] = (self.documents1[ID], \
                               self.documents2[ID], self.calcsimscore(ID))
        return annotTuples

    def generatemixed(self,tag,quantity=None):
        if self.noMix:
            return
        if quantity is None:
            quantity = len(self.corpus)
        global globalindex
        counter = 0
        keys1 = self.corpus.keys() # get corpus keys
        keys2 = keys1[:] # create copy
        while counter < quantity:
            random1 = random.randint(0,len(keys1)-1)
            random2 = random.randint(0,len(keys1)-1)
            key1 = keys1[random1]
            key2 = keys2[random2]
            if key1 == key2: continue
            ID = tag+str(globalindex)+'m'
            self.mixedIDs.append(ID)
            self.mixedcorpus[ID] = (self.corpus[key1][0],
                                    self.corpus[key2][1], 0.0)
            del keys1[random1], keys2[random2]
            globalindex += 1
            counter += 1

class lapatacorp(corpusbuilder):
    """Class for generating sections derived from Lapata corpus.
    """
    __annotations_A = {}
    __annotations_C = {}
    tokens1 = {}
    tokens2 = {}

    def __init__(self,rawfile1,rawfile2,fileA,fileC,tag=""):
        corpusbuilder.__init__(self)
```

```
        if (rawfile1 == None) or (rawfile2 == None) or \
                                (fileA == None) or (fileC == None):
            raise Exception, "Missing filenames in constructor args."
        if len(tag)>0: self.__tag = "Paraphrase-"+tag
        else: self.__tag = "Paraphrase"

        # store tokens for each sentence one of the paraphrase
        self.documents1, self.documents2, self.IDs, self.tokens1, self.tokens2 \
                                    = self.getdocs(rawfile1,rawfile2)

        # store alignment annotations for annotators
        self.__annotations_A, self.__annotations_C = \
                                self.annotparser(fileA, fileC, self.IDs)
        self.corpus = self.scoreCorpus()
        self.generatemixed(self.__tag)

    def getdocs(self, rawfile1, rawfile2):
        global globalindex
        rawdocs1, rawIDs = self.rawparser(rawfile1)
        rawdocs2 = self.rawparser(rawfile2)[0]
        docs1 = {}
        docs2 = {}
        tokens1 = {}
        tokens2 = {}
        IDs = []
        for ID in rawIDs:
            newID = self.__tag+str(globalindex)
            tokens1[newID] = rawdocs1[ID]
            tokens2[newID] = rawdocs2[ID]
            docs1[newID] = " ".join(rawdocs1[ID])
            docs2[newID] = " ".join(rawdocs2[ID])
            IDs.append(self.__tag+str(globalindex))
            globalindex += 1
        return docs1, docs2, IDs, tokens1, tokens2

    def rawparser(self,rawfile):
        sentences = {}
        orderedIDs = []
        rawfile.seek(0)
        for line in rawfile:
            #group up tags and raw sentence
            parts = re.search(r"(<s snum=[0-9]+>) ([^<]*) (</s>)",line)
             #strip away tags
            sentence = parts.group(2)
             # extract id number from leading <s...> tag
            id = int(re.search(r"[0-9]+",parts.group(1)).group(0))
            orderedIDs.append(id)
            sentences[id] = sentence.split()
        return (sentences,orderedIDs)

    def annotparser(self,fileA,fileC,IDs):
        annotA = {}
        annotC = {}
        fileA.seek(0)
        fileC.seek(0)
        for line in fileA:
            parts = re.search(r"([0-9]+) ([0-9]+) ([0-9]+) ([SP])",line)
```

```
            key = IDs[int(parts.group(1))]
            align1 = int(parts.group(2))
            align2 = int(parts.group(3))
            note = parts.group(4)
            try:
                annotA[key][(align1,align2)] = note
            # this should happen less than 10% of the time, so we'll use try
            except KeyError:
                annotA[key] = {(align1,align2):note}
        for line in fileC:
            parts = re.search(r"([0-9]+) ([0-9]+) ([0-9]+) ([SP])",line)
            key = IDs[int(parts.group(1))]
            align1 = int(parts.group(2))
            align2 = int(parts.group(3))
            note = parts.group(4)
            try:
                annotC[key][(align1,align2)] = note
            # this should happen less than 10% of the time, so we'll use try
            except KeyError:
                annotC[key] = {(align1,align2):note}
        for id in IDs:
            for alignpair in annotA[id]:
                if alignpair not in annotC[id]: annotC[id][alignpair] = '-'
            for alignpair in annotC[id]:
                if alignpair not in annotA[id]: annotA[id][alignpair] = '-'
        return (annotA,annotC)

    def calcsimscore(self,index):
        """ Returns the F-measure of inter-annotator agreement.
        """
        As = 0
        Cs = 0
        AsIntCp = 0
        ApIntCs = 0

        for pair in self.__annotations_A[index]:
            word1index, word2index = pair
            noteA = self.__annotations_A[index][pair]
            noteC = self.__annotations_C[index][pair]
            word1 = self.tokens1[index][word1index-1].lower()
            word2 = self.tokens2[index][word2index-1].lower()

            if not word1  == word2:
                if noteA == 'S':
                    As += 1
                    if (noteC == 'P') or (noteC == 'S'):
                        AsIntCp += 1
                if noteC == 'S':
                    Cs += 1
                    if (noteA == 'P') or (noteA == 'S'):
                        ApIntCs += 1

        if As > 0: prec = AsIntCp / As
        else: prec = 1
        if Cs > 0: rec = ApIntCs / Cs
        else: rec = 1
        if prec+rec > 0: return (2*prec*rec)/(prec+rec)
```

```
        else: return 0

class wikicorp(corpusbuilder):

    def __init__(self, urllist,tag=""):
        corpusbuilder.__init__(self)
        global globalindex
        if len(tag)>0: self.__tag = "Wikipedia-"+tag
        else: self.__tag = "Wikipedia"
        urllist.seek(0)
        for url in urllist:
            try:
                if url.find(r"&printable=yes"):
                    html_eng = self.getHTML(self.getprintURL(url))
                else:
                    html_eng = self.getHTML(url)
            except:
                sys.stderr.write("Couldn't retrieve "+url+"\nSkipping...\n")
                continue
            try:
                simplepatt = r'(http://simple\.wikipedia\.org/'+ \
                                 r'wiki/[^/<>\s"]+[/]{0,1})'
                urlsimple = re.search(simplepatt,html_eng).group(0).strip('\n')
            except AttributeError:
                sys.stderr.write("No Simple English version of "+ \
                                 url+"\nSkipping...\n")
                continue
            try:
                html_simp = self.getHTML(self.getprintURL(urlsimple))
            except:
                sys.stderr.write("Couldn't retrieve "+ \
                                 urlsimple+"\nSkipping...\n")
                continue
            try:
                text_eng = clean_html(html_eng)
                text_simp = clean_html(html_simp)
                self.documents1[self.__tag+str(globalindex)] = \
                                                self.getarticlebody(text_eng)
                self.documents2[self.__tag+str(globalindex)] = \
                                                self.getarticlebody(text_simp)
            except:
                sys.stderr.write("Problem came up while cleaning up "+ \
                                 url+"\nSkipping...\n")
                continue
            self.IDs.append(self.__tag+str(globalindex))
            globalindex +=1
        self.corpus = self.scoreCorpus()
        self.generatemixed(self.__tag)


    def getHTML(self, url):
        opener = urllib2.build_opener()
        opener.addheaders = [('User-agent', 'Mozilla/5.0')]
        infile = opener.open(url)
        page = infile.read()
        return page
```

```
    def getprintURL(self, url):
        parts = re.search(r"(http://)([a-zA-Z]+)(.wikipedia.org/wiki/)(.*)",url)
        formattedurl = "http://"+parts.group(2)+ \
                        ".wikipedia.org/w/index.php?title="+ \
                        parts.group(4)+"&printable=yes"
        return formattedurl

    def getarticlebody(self,raw):
        # find start of boilerplate text
        boilerplate = raw.find(r'Retrieved from "http://')
        # remove boilerplate text
        cleantext = raw[:boilerplate]
        # remove large chunks of whitespace
        cleantext = re.sub(r"[ \t]{2}[ \t]*"," ",cleantext)
        # bunch up newlines
        cleantext = re.sub(r"[ \t]+\n","\n",cleantext)
        # reduce large chunks of newlines
        cleantext = re.sub(r"\n\n\n+","\n\n",cleantext)
        return cleantext

class papercorp(corpusbuilder):
    def __init__(self,folder,tag=""):
        corpusbuilder.__init__(self)
        global globalindex
        rawpdfs = []

        if len(tag)>0: self.__tag = "Abstracts-"+tag
        else: self.__tag = "Abstracts"

        # These get absolute path (computationally cheap, so let's skip checks)
        folder = os.path.expanduser(folder)
        folder = os.path.expandvars(folder)
        folder = os.path.abspath(folder)

        if not os.path.exists(folder):
            raise Exception, folder+" does not exist!"

        pdflist = glob.glob(folder+"/*.pdf")

        for pdf in pdflist:
            try:
                raw = self.getrawtxt(pdf)
            except:
                sys.stderr.write("\nSomething went wrong while trying to " + \
                                 "open "+pdf+". Perhaps it is write "+ \
                                 "protected? Skipping...\n")
                continue
            rawpdfs.append(raw)

        for rawtext in rawpdfs:
            try:
                abstract, body = self.chopabstract(rawtext)
            except:
                sys.stderr.write("\nSomething went wrong trying to chop up "+ \
                                 "document.\nHere is a "+ \
                                 "sample of the first few lines:\n"+ \
                                 "=============\n" + rawtext[:250]+ \
```

```
                                "\n=============\nSkipping...\n")

                continue
            self.documents1[self.__tag+str(globalindex)] = abstract
            self.documents2[self.__tag+str(globalindex)] = body
            self.IDs.append(self.__tag+str(globalindex))
            globalindex += 1

        self.corpus = self.scoreCorpus()
        self.generatemixed(self.__tag)

    def getrawtxt(self,pdf):
        textFN = pdf[:pdf.find('.pdf')]+".txt"
        retcode = call(["/usr/local/bin/pdftotext", "-nopgbrk", \
                        "-raw",pdf,textFN])
        if retcode is not 0: # pdftotxt failed to process pdf
            raise Exception # This will be handled in __init__
        try:
            textfile = open(textFN)
        except:
            sys.stderr.write("Something went really wrong. Check pdftotext.\n")
            sys.stderr.write("Problem happened with "+pdf+"\nSkipping...\n")
            sys.exit()
        try:
            raw = "".join(textfile.readlines())
        except:
            raise Exception # This will be handled in __init__
        finally: # to avoid potential memory leaks if this messes up
            textfile.close()
            os.remove(textFN) # clean up generated text file
        return raw

    def chopabstract(self,raw):
        if "introduction" in raw[:2500].lower():
            regex = r"(abstract.*?)(introduction.*)"
        elif "keyword" in raw[:2500].lower():
            regex = r"(keyword.*?)(introduction.*)"
        else: raise Exception # introduction/keywords not found

        slicer = re.compile(regex, re.I|re.DOTALL)
        parts = slicer.search(raw)
        if parts is None:
            regex = regex = r"(.*?)(introduction.*)"
            slicer = re.compile(regex, re.I|re.DOTALL)
            parts = slicer.search(raw)
        abstract = self.cleantext(parts.group(1))
        body = self.cleantext(parts.group(2))
        return abstract, body

    def cleantext(self,text):
        cleantext = text.replace('-\n ','') # fix split words
        cleantext = cleantext.replace('-\n','') # ditto
        # remove newlines in the middle of sentences
        cleantext = re.sub(r"(?<=[\w:.,-?!;()])\n(?=[\w])", ' ', cleantext)
        return cleantext

class editcorp(corpusbuilder):
```

```
    noMix = True
    def __init__(self,startindex=0,size=500):
        corpusbuilder.__init__(self)
        global globalindex
        self.sentences = brown.sents()[startindex:startindex+size]
        for sentence in self.sentences:
            sent1 = " ".join(sentence)
            sent2, edits = self.randomedits(sent1)
            score = 1 - (edits/len(sent1))
            ID = "Edits" + str(globalindex)
            self.IDs.append(ID)
            self.corpus[ID] = (sent1, sent2, score)
            globalindex += 1

    def randomedits(self,sentence):
        edits = 0
        randindex = lambda x: random.randint(0,len(x)-1)
        maxchanges = randindex(sentence)

        for i in range(0,maxchanges):
            edit_type = random.randint(1,3)
            try: edit_loc = randindex(sentence)
            except:
                print sentence
                sys.exit()
            if edit_type is 1:
                sentence = self.morph(sentence,edit_loc)
            elif edit_type is 2:
                sentence = self.delete(sentence,edit_loc)
            elif edit_type is 3:
                destination = randindex(sentence)
                while destination is edit_loc:
                    destination = randindex(sentence)
                sentence = self.move(sentence,edit_loc,destination)
            edits += 1
        return (sentence,edits)

    def morph(self,sent,loc):
        target = "1234567890abcdefghijklmnopqrstuvw"\
                 "xyzABCDEFGHIJKLMNOPQRSTUVWXYZ,.;:?!"
        original = sent[loc]
        change = target[random.randint(0,len(target)-1)]
        while change is original:
            change = target[random.randint(0,len(target)-1)]
        return sent[:loc]+change+sent[loc+1:]

    def delete(self,sent,loc):
        return sent[:loc]+sent[loc+1:]

    def move(self,sent,loc,dest):
        character = sent[loc]
        sent = self.delete(sent, loc)
        if dest > loc: dest -= 1
        return sent[:dest]+character+sent[dest:]

class themecorp(corpusbuilder):
    noMix = True
```

```
    text_chunks = {}
    cats = ['adventure','editorial','romance','religion','science_fiction']
    dualcats = {'adventure':'editorial',
                'editorial':'romance',
                'romance':'science_fiction',
                'religion':'adventure',
                'science_fiction':'religion'}

    def __init__(self,startindex=0,set_size=3,set_no=50):
        corpusbuilder.__init__(self)
        self.__tag = "Themes"
        global globalindex
        for category in self.cats:
            index = startindex
            self.text_chunks[category] = []
            while index-startindex < (set_no*set_size*2):
                chunk = ""
                for i in range (0,set_size):
                    para = " ".join(reduce(lambda x,y:x+y,
                                    brown.paras(categories=category)[index+i]))
                    chunk += para
                self.text_chunks[category].append(chunk)
                index += set_size

        for category in self.cats:
            for i in range(0,len(self.text_chunks[category]),2):
                sent1 = self.text_chunks[category][i]
                sent2 = self.text_chunks[category][i+1]
                ID = self.__tag+str(globalindex)
                self.IDs.append(ID)
                self.corpus[ID] = (sent1,sent2,1.0)
                globalindex += 1

        for category in self.cats:
            for i in range(0,set_no):
                sent1 = self.text_chunks[category][i]
                sent2 = self.text_chunks[self.dualcats[category]][i]
                ID = self.__tag+str(globalindex)+"m"
                self.mixedIDs.append(ID)
                self.mixedcorpus[ID] = (sent1,sent2,0.0)
                globalindex += 1

class POSswitchcorp(corpusbuilder):
    def __init__(self,startindex=1500,size=500):
        corpusbuilder.__init__(self)
        global globalindex
        self.__tag = "POSswitch"
        self.tagged_sents = brown.tagged_sents()[startindex:startindex+size]
        POScatwords = {}
        for pair in reduce(lambda x,y:x+y, self.tagged_sents):
            word, tag = pair
            try: POScatwords[tag].append(word)
            except: POScatwords[tag] = [word]
        for tag in POScatwords:
            POScatwords[tag] = list(set(POScatwords[tag]))
        for tagged_sent in self.tagged_sents:
            dual_sent = tagged_sent[:] # create working copy
```

```
        for i in range(0,random.randint(1,len(tagged_sent))):
            loc = random.randint(0,len(tagged_sent)-1)
            type = dual_sent[loc][1]
            wordlist = POScatwords[type]
            subs_word = wordlist[random.randint(0,len(wordlist)-1)]
            if dual_sent[loc][0].islower():
                subs_word = subs_word[0].lower() + subs_word[1:]
            else: subs_word = subs_word[0].upper() + subs_word[1:]
            dual_sent[loc] = (subs_word,type)
        tagged_sent = " ".join(map(lambda x:x[0],tagged_sent))
        dual_sent = " ".join(map(lambda x:x[0],dual_sent))
        ID = self.__tag+str(globalindex)
        self.IDs.append(ID)
        self.documents1[ID] = tagged_sent
        self.documents2[ID] = dual_sent
        globalindex += 1
    self.corpus = self.scoreCorpus()
    self.generatemixed(self.__tag)
```

B.2 buildcorpus.py

```
#!/usr/local/bin/python
"""A script for using building experimental corpus.

Written by Edward Grefenstette, University of Oxford, 2009.
email: <edward.grefenstette@balliol.ox.ac.uk>
Submitted towards completion of MSc Computer Science.
"""

import corpusbuilder
import cPickle as pickle

datadir = r"/Users/Edward/Dropbox/MSc CS/MSc Project/data"

MTCdocs = map(lambda x:datadir+x,
               [r"/LapataCorp/mtc.common.one", r"/LapataCorp/mtc.common.two",
                r"/LapataCorp/mtcAx.align", r"/LapataCorp/mtcCx.align"])
newsdocs = map(lambda x:datadir+x,
               [r"/LapataCorp/news.common.one", r"/LapataCorp/news.common.two",
                r"/LapataCorp/newsAx.align", r"/LapataCorp/newsCx.align"])
noveldocs = map(lambda x:datadir+x,
               [r"/LapataCorp/novels.common.one",
                r"/LapataCorp/novels.common.two",
                r"/LapataCorp/novelsAx.align", r"/LapataCorp/novelsCx.align"])

wikiloc = datadir+"/wiki_articles.txt"

paperdir = datadir+"/papers"

## Begin Corpus Construction
print "Beginning corpus construction."
IDs = []
corpus = {}
```

```
## Build Syntactic toy corpora
print "Building Syntactic Toy corpora."
builder = corpusbuilder.editcorp()
IDs += builder.IDs
corpus.update(builder.corpus)
IDs += builder.mixedIDs
corpus.update(builder.mixedcorpus)

builder = corpusbuilder.POSswitchcorp()
IDs += builder.IDs
corpus.update(builder.corpus)
IDs += builder.mixedIDs
corpus.update(builder.mixedcorpus)

## Build Theme corpus
print "Building Theme corpus."
builder = corpusbuilder.themecorp()
IDs += builder.IDs
corpus.update(builder.corpus)
IDs += builder.mixedIDs
corpus.update(builder.mixedcorpus)

## Build MTC corpus
print "Building MTC corpus."
sents1 = open(MTCdocs[0])
sents2 = open(MTCdocs[1])
annot1 = open(MTCdocs[2])
annot2 = open(MTCdocs[3])

builder = corpusbuilder.lapatacorp(sents1,sents2,annot1,annot2,"MTC")
IDs += builder.IDs
corpus.update(builder.corpus)
IDs += builder.mixedIDs
corpus.update(builder.mixedcorpus)

sents1.close()
sents2.close()
annot1.close()
annot2.close()


## Build News corpus
print "Building News corpus."
sents1 = open(newsdocs[0])
sents2 = open(newsdocs[1])
annot1 = open(newsdocs[2])
annot2 = open(newsdocs[3])

builder = corpusbuilder.lapatacorp(sents1,sents2,annot1,annot2,"News")
IDs += builder.IDs
corpus.update(builder.corpus)
IDs += builder.mixedIDs
corpus.update(builder.mixedcorpus)

sents1.close()
sents2.close()
```

```
annot1.close()
annot2.close()

IDs += builder.IDs
corpus.update(builder.corpus)
IDs += builder.mixedIDs
corpus.update(builder.mixedcorpus)

## Build Novel corpus
print "Building Novel corpus."
sents1 = open(noveldocs[0])
sents2 = open(noveldocs[1])
annot1 = open(noveldocs[2])
annot2 = open(noveldocs[3])

builder = corpusbuilder.lapatacorp(sents1,sents2,annot1,annot2,"Novel")
IDs += builder.IDs
corpus.update(builder.corpus)
IDs += builder.mixedIDs
corpus.update(builder.mixedcorpus)

sents1.close()
sents2.close()
annot1.close()
annot2.close()

## Build Wiki Corpus
print "Building Wikipedia corpus."
urllist = open(wikiloc)
builder = corpusbuilder.wikicorp(urllist)
IDs += builder.IDs
corpus.update(builder.corpus)
urllist.close()
IDs += builder.mixedIDs
corpus.update(builder.mixedcorpus)

## Build Paper Corpus
print "Building Abstract-Article corpus."
builder = corpusbuilder.papercorp(paperdir)
IDs += builder.IDs
corpus.update(builder.corpus)
IDs += builder.mixedIDs
corpus.update(builder.mixedcorpus)

## Outputting to pickle file
print "Writing corpus to pickle file."
outputloc = r"/Users/Edward/Dropbox/MSc CS/MSc Project/corpus/docsimcorp.pkl"
outputF = open(outputloc,"w")
pickle.dump((corpus,IDs), outputF)
outputF.close()
```

B.3 prepcorpus.py

```
#!/usr/local/bin/python
"""A script for pre-processing a corpus.

Written by Edward Grefenstette, University of Oxford, 2009.
email: <edward.grefenstette@balliol.ox.ac.uk>
Submitted towards completion of MSc Computer Science.
"""

def main():
    try:
        corpusfile = open(args[0],"r")
    except:
        sys.stderr.write("Could not open file "+args[0]+".\n")
        sys.exit()

    corpus, IDs = pickle.load(corpusfile)
    eval_data = preprocessCorpus(corpus)
    corpusfile.close()

    if not options.quiet: print "Saving pre-processed corpus to" +\
                            args[1]+" and quitting."
    prepfile = open(args[1],"w")
    pickle.dump(eval_data, prepfile)
    prepfile.close()
    return

def preprocessCorpus(corpus):
    if not options.quiet: print "==Pre-processing corpus=="
    if not options.quiet: print "=Formatting corpus="
    rawdata = format(corpus,options.quiet)
    if not options.quiet:
        print "=Tagging corpus="
        print "(This may take over an hour.)"
    eval_data = tagcorpus(rawdata,options.quiet)
    return eval_data

def format(corpus, quiet=False):
    eval_data = {}
    next = 10
    counter = 0
    if not quiet:
        sys.stdout.write("Progress: 0%")
        sys.stdout.flush()
    for ID in corpus:
        tokens1 =  word_tokenize(corpus[ID][0])
        tokens2 =  word_tokenize(corpus[ID][1])

        # Normalisation step
        string1 = " ".join(tokens1)
        string2 = " ".join(tokens2)

        gold_standard = corpus[ID][2]
        eval_data[ID] = (string1,string2,tokens1,
                            tokens2,gold_standard)
```

```
        # UI friendliness
        counter += 1
        if not quiet:
            while (counter*100/len(corpus)) >= next:
                if (counter*100/len(corpus)) < 100:
                    sys.stdout.write("-"+str(next)+"%")
                    sys.stdout.flush()
                else:
                    print "-100%"
                next += 10
    return eval_data

def tagcorpus(rawdata, quiet=False):
    tagged_data = {}
    next = 1
    counter = 0
    if not quiet:
        sys.stdout.write("Progress: 0% ")
        sys.stdout.flush()
    for ID in rawdata:
        tokens1, tokens2 = rawdata[ID][2:4]
        tagged1 = pos_tag(tokens1)
        tagged2 = pos_tag(tokens2)
        tagged_data[ID] = rawdata[ID]+(tagged1,tagged2)
        counter += 1
        if not quiet:
            while (counter*100/len(rawdata)) >= next:
                if (counter*100/len(rawdata)) < 100:
                    sys.stdout.write(str(next)+"% ")
                    sys.stdout.flush()
                else:
                    print "100%"
                next += 1
    return tagged_data

if __name__ == "__main__":
    import sys
    from optparse import OptionParser

    usage = "%prog -q CORPUS PREPFILE"
    description = ""

    parser = OptionParser(usage = usage, description = description)

    parser.add_option("-q", "--quiet", dest="quiet", action="store_true",
                      default = False, help="no output to stdin.")

    (options, args) = parser.parse_args()

    if len(args) < 1:
        parser.print_help()
        sys.exit()

    # Import Psyco if available
    try:
        if not options.quiet: print "Importing psyco..."
```

```
        import psyco
        psyco.full()
    except ImportError:
        pass

    import cPickle as pickle
    from corpusevaluation import *
    from nltk import word_tokenize, pos_tag

    main()
```

B.4 corpusevaluation.py

```
"""A collection of metric evaluation classes.

Written by Edward Grefenstette, University of Oxford, 2009.
email: <edward.grefenstette@balliol.ox.ac.uk>
Submitted towards completion of MSc Computer Science.
"""

from __future__ import division
from Levenshtein import distance, hamming, jaro, ratio
import nltk
from nltk import word_tokenize
from nltk.metrics import edit_distance, jaccard_distance, masi_distance
from nltk.corpus import wordnet as wn
from nltk.corpus import wordnet_ic as wnic
import os, sys
from subprocess import Popen, PIPE
import random
import shutil
import re
from math import ceil

# General evaluator classes

class evaluator(object):
    EVALERROR = 0
    eval_data = {}
    scores = {}

    def __init__(self, corpus = None, eval_data = None, quiet = False):
        try:
            import psyco
            psyco.full()
        except ImportError:
            pass
        if corpus is not None:
            if not quiet: print "=Formatting corpus="
            self.format(corpus, quiet)
        elif eval_data is not None:
            self.eval_data = eval_data
        else:
```

```
            raise Exception, "Class given nothing to evaluate!"
        if not quiet: print "=Scoring corpus="
        self.scores = self.getscore(quiet)

    def format(self, corpus, quiet=False):
        next = 10
        counter = 0
        if not quiet:
            sys.stdout.write("Progress: 0%")
            sys.stdout.flush()
        for ID in corpus:
            tokens1 = word_tokenize(corpus[ID][0])
            tokens2 = word_tokenize(corpus[ID][1])

            # Normalisation step
            string1 = " ".join(tokens1)
            string2 = " ".join(tokens2)

            gold_standard = corpus[ID][2]
            self.eval_data[ID] = (string1,string2,tokens1,
                                  tokens2,gold_standard)
            counter += 1
            if not quiet:
                while (counter*100/len(corpus)) >= next:
                    if (counter*100/len(corpus)) < 100:
                        sys.stdout.write("-"+str(next)+"%")
                        sys.stdout.flush()
                    else:
                        print "-100%"
                    next += 10

    def getscore(self, quiet = False):
        next = 1
        counter = 0
        scores = {}
        if not quiet:
            sys.stdout.write("Progress: 0%")
            sys.stdout.flush()
        for ID in self.eval_data:
            gold_standard = self.eval_data[ID][4]
            linescore = self.scoreline(self.eval_data[ID])
            difference = abs(gold_standard - linescore)
            scores[ID] = 1 - difference
            counter += 1
            if not quiet:
                while (counter*100/len(self.eval_data)) >= next:
                    if (counter*100/len(self.eval_data)) < 100:
                        sys.stdout.write(" "+str(next)+"%")
                        sys.stdout.flush()
                    else:
                        print " 100%"
                    next += 1
        return scores

    def scoreline(self,line):
        raise Exception, "(Parent Class Warning) No implemented functions."
```

```
class TAGevaluator(evaluator):
    tagged_data = {}
    stopwords = nltk.corpus.stopwords.words('english')

    def __init__(self,corpus = None, eval_data=None, quiet=False):
        if corpus is not None:
            if not quiet: print "=Formatting corpus="
            self.format(corpus,quiet)
            if not quiet:
                print "=Tagging corpus="
                print "(This may take over an hour for large corpora.)"
            self.tagged_data = self.tagcorpus(self.eval_data,quiet)
            self.eval_data = self.tagged_data
        elif eval_data is not None:
            self.eval_data = eval_data
        else:
            raise Exception, "Class given nothing to evaluate!"
        if not quiet: print "=Scoring corpus="
        self.scores = self.getscore()

    def tagcorpus(self,rawdata, quiet=False):
        next = 1
        counter = 0
        if not quiet:
            sys.stdout.write("Progress: 0%")
            sys.stdout.flush()
        for ID in rawdata:
            tokens1, tokens2 = rawdata[ID][2:4]
            tagged1 = nltk.pos_tag(tokens1)
            tagged2 = nltk.pos_tag(tokens2)
            self.tagged_data[ID] = rawdata[ID]+(tagged1,tagged2)
            counter += 1
            if not quiet:
                while (counter*100/len(rawdata)) >= next:
                    if (counter*100/len(rawdata)) < 100:
                        sys.stdout.write(" "+str(next)+"%")
                        sys.stdout.flush()
                    else:
                        print " 100%"
                    next += 1

class WNevaluator(TAGevaluator):
    metric = None
    ic = None

    def scoreline(self,line):
        scores = []
        best = {}
        if self.metric is None:
            raise Exception, "Use one of WNevaluator' children!"

        nouns1,nouns2,verbs1,verbs2 = self.getgroups(line)

        synd1 = dict([[x,wn.synsets(x,'n')]
                      for x in set(map(lambda x:x.lower(),nouns1))])
        synd2 = dict([[x,wn.synsets(x,'n')]
                      for x in set(map(lambda x:x.lower(),nouns2))])
```

```
        pairs = [(x,y) for x in synd1.keys() for y in synd2.keys()]

        for pair in pairs:
            key1, key2 = pair
            if (key2,key1,'n') in best:
                best[pair+('n',)] = best[(key2,key1,'n')]
                continue
            best[pair+('n',)] = 0.0
            if len(synd1[key1]) == 0 or len(synd2[key2]) == 0: continue
            for syn1 in synd1[key1]:
                for syn2 in synd2[key2]:
                    try: mark = self.metric(syn1,syn2,self.ic)
                    except: mark = 0.0
                    if mark < 0: mark = 0
                    if mark > best[pair+('n',)]: best[pair+('n',)] = mark

        synd1 = dict([[x,wn.synsets(x,'v')]
                      for x in set(map(lambda x:x.lower(),verbs1))])
        synd2 = dict([[x,wn.synsets(x,'v')]
                      for x in set(map(lambda x:x.lower(),verbs2))])

        pairs = [(x,y) for x in synd1.keys() for y in synd2.keys()]

        for pair in pairs:
            key1, key2 = pair
            if (key2,key1,'v') in best:
                best[pair+('v',)] = best[(key2,key1,'v')]
                continue
            best[pair+('v',)] = 0.0
            if len(synd1[key1]) == 0 or len(synd2[key2]) == 0: continue
            for syn1 in synd1[key1]:
                for syn2 in synd2[key2]:
                    try: mark = self.metric(syn1,syn2,self.ic)
                    except: mark = 0.0
                    if mark < 0: mark = 0
                    if mark > best[pair+('v',)]: best[pair+('v',)] = mark

        runningscore = 0.0
        runningcount = 0
        scores = []
        for pair in best: scores.append(best[pair])
        scores.sort()
        scores.reverse()
        best10p = scores[:int(ceil(len(best)/10))]
        if len(best10p) == 0:
            return 0.0
        else:
            return sum(best10p) / len(best10p)

    def getgroups(self,line):
        # From Penn Treebank tagset
        nountags = ['FW','NN','NNS','NP','NPS']
        verbtags = ['FW','VB','VBD','VBG','VBN','VBP','VBZ']
        nouns1 = []
        nouns2 = []
        verbs1 = []
```

```
        verbs2 = []
        tagged1, tagged2 = line[5:7]
        for pair in tagged1:
            word = pair[0].lower()
            if pair[1] in nountags: nouns1.append(word)
            if pair[1] in verbtags: verbs1.append(word)
        for pair in tagged2:
            word = pair[0].lower()
            if pair[1] in nountags: nouns2.append(word)
            if pair[1] in verbtags: verbs2.append(word)
        nouns1 = [noun for noun in nouns1 if noun not in self.stopwords]
        nouns2 = [noun for noun in nouns2 if noun not in self.stopwords]
        verbs1 = [verb for verb in verbs1 if verb not in self.stopwords]
        verbs2 = [verb for verb in verbs2 if verb not in self.stopwords]
        return (nouns1,nouns2,verbs1,verbs2)

# Actual evaluation classes

class wordcounteval(evaluator):
    def scoreline(self,line):
        string1, string2, tokens1, tokens2 = line[:4]
        maxlength = max(len(tokens1),len(tokens2))
        diff = abs(len(tokens1) - len(tokens2))
        score = 1 - diff/maxlength
        return score

class charcounteval(evaluator):
    def scoreline(self,line):
        string1, string2 = line[:2]
        maxlength = max(len(string1),len(string2))
        diff = abs(len(string1) - len(string2))
        score = 1 - diff/maxlength
        return score

class leveneval(evaluator):
    def scoreline(self,line):
        string1, string2 = line[:2]
        maxlength = max(len(string1),len(string2))
        score = 1.0 - distance(string1,string2)/maxlength
        if score < 0: score = 0.0 # Sanity check. Should never happen.
        return score

class jaroeval(evaluator):
    def scoreline(self,line):
        string1, string2 = line[:2]
        return jaro(string1,string2)

class ratioeval(evaluator):
    def scoreline(self,line):
        string1, string2 = line[:2]
        return ratio(string1,string2)

class jaccardeval(evaluator):
    def scoreline(self,line):
        tokens1, tokens2 = line[2:4]
        set1 = set(tokens1)
        set2 = set(tokens2)
```

```
        score = 1.0 - jaccard_distance(set1, set2)
        return score

class masieval(evaluator):
    def scoreline(self,line):
        tokens1, tokens2 = line[2:4]
        set1 = set(tokens1)
        set2 = set(tokens2)
        score = 1.0 - masi_distance(set1, set2)
        return score

class pathsimeval(WNevaluator):
    metric = wn.path_similarity

class WUPeval(WNevaluator):
    metric = wn.wup_similarity

class LINeval(WNevaluator):
    metric = wn.lin_similarity
    ic = wnic.ic('ic-bnc.dat') # Using British National Corpus as IC

class SemVecteval(evaluator):
    def getscore(self, quiet = False):
        pkgpath = os.path.expandvars("$SEMVECT")
        os.chdir(pkgpath)
        folderpath = "./folder" + str(random.randint(1,1000))
        while(os.path.exists(folderpath)):
            folderpath = "./folder" + str(random.randint(1,1000))
        os.mkdir(folderpath)

        indexpath = folderpath+"/fileindex"
        fileindex = open(indexpath,"w")

        for ID in self.eval_data:
            fileindex.write(ID+"\n")
            fileOne = open(folderpath+"/"+ID+".one","w")
            fileindex.write(folderpath+"/"+ID+".one\n")
            fileTwo = open(folderpath+"/"+ID+".two","w")
            fileindex.write(folderpath+"/"+ID+".two\n")
            fileOne.write(self.eval_data[ID][0])
            fileTwo.write(self.eval_data[ID][1])
            fileOne.close()
            fileTwo.close()

        fileindex.close()
        scorefile = open(folderpath+"/scores","w")
        scorefile.close()

        if not os.path.exists(pkgpath):
            sys.stderr.write("Couldn't find semantic vectors package! " +
                             "Use ./scripts/evaluatemetrics or set the "+
                             "$SEMVECT environment variable to the folder "+
                             "containing the Semantic Vectors Evaluator "+
                             "package. Quitting.\n")
            sys.exit()

        arglist =   "java -Xmx1024m -cp " + \
```

```
                    os.path.expandvars("$CLASSPATH") + \
                    " SemanticVectorsEvaluator " + folderpath + \
                    " " + str(len(self.eval_data))

        SemVectPackage = Popen(arglist, shell=True)
        SemVectPackage.wait()

        semscores = {}
        scorefile = open(folderpath+"/scores")

        for line in scorefile:
            line = line.strip('\n')
            parts = re.search(r"(\S*) (\S*)", line)
            semscores[parts.group(1)] = float(parts.group(2))

        scorefile.close()

        shutil.rmtree(folderpath)
        os.chdir("..")

        scores = {}

        for ID in self.eval_data:
            gold_standard = self.eval_data[ID][4]
            linescore = semscores[ID]
            difference = abs(gold_standard - linescore)
            scores[ID] = 1.0 - difference

        return scores

class BLEUeval(evaluator):
    def scoreline(self,line):
        doc1, doc2 = line[:2]
        doc1 = doc1.replace('\n', ' ')
        doc2 = doc2.replace('\n', ' ')
        doc1f = open('./doc1f','w')
        doc2f = open('./doc2f','w')
        doc1f.write(doc1+"\n*")
        doc2f.write(doc2+"\n*")
        doc1f.close()
        doc2f.close()
        arglist = './tools/bleu ./doc1f ./doc2f'
        BLEU = Popen(arglist, shell=True, stdout = PIPE)
        BLEU.wait()
        output = BLEU.stdout.read()
        outsearch = re.search(r"(?<=Bleu = )([01]\.[0-9]*)", output)
        try:
            score = float(outsearch.group(0))
        except:
            score = 0.0
        os.remove('./doc1f')
        os.remove('./doc2f')
        return score

class randomeval(evaluator):
    def scoreline(self,line):
        return random.uniform(0.0,1.0)
```

B.5 semanticvectors.java

```
/* A collection of corpus generating classes.
 *
 * Based on class written by Pascal Combescot.
 * Modified by Edward Grefenstette, University of Oxford, 2009.
 * email: <edward.grefenstette@balliol.ox.ac.uk>
 * Submitted towards completion of MSc Computer Science.
*/

import java.io.*;
//import java.util.*;

import pitt.search.semanticvectors.*;
import englishStopWords.EnglishStopWordsFilter;

public class SemanticVectorsEvaluator {

        /**
         * @param args
         * @throws Exception
         */
        public static void main(String[] args) throws Exception {

                //### Build the vector database ###
                VectorStoreRAM vecReaderRAM = new VectorStoreRAM();
                vecReaderRAM.InitFromFile(Flags.queryvectorfile);

                EnglishStopWordsFilter filter = new EnglishStopWordsFilter();
                float score;



                String folderAddress = args[0];

                String indexAddress = folderAddress + "/fileindex";
                BufferedReader indexF;
                indexF = new BufferedReader(new FileReader(indexAddress));
                String ID = indexF.readLine();
                String fileAddress1 = indexF.readLine();
                String fileAddress2 = indexF.readLine();

                String string1="", string2="";
                BufferedReader file;
                String line;

                File outputF = new File(folderAddress+"/scores");
                PrintWriter out = new PrintWriter(new FileWriter(outputF));
                String output = "";

                int datalen = Integer.parseInt(args[1]);
```

```
        int counter = 0;
        int next = 1;

        System.out.print("Progress: 0%");
System.out.flush();

        while(ID!=null){
                ID = ID.replaceAll("\n", "");
                fileAddress1 = fileAddress1.replaceAll("\n", "");
                fileAddress2 = fileAddress2.replaceAll("\n", "");

                string1 = "";
                string2 = "";


                file = new BufferedReader(new FileReader(fileAddress1));
                line = file.readLine();
                while(line!=null){
                        string1 = string1 + line;
                        line = file.readLine();
                }
                file.close();

                file = new BufferedReader(new FileReader(fileAddress2));
                line = file.readLine();
                while(line!=null){
                        string2 = string2 + " " + line;
                        line = file.readLine();
                }
                file.close();

                try {
                        score = getScore(filter.stringToStringArray(string1),
                                        filter.stringToStringArray(string2),
                                        vecReaderRAM, false);
                }
                catch (Exception e) {
                        score = 0;
                }


                output = ID + " " + Float.toString(score) + "\n";
                out.print(output);

                counter++;

                while ((counter*100/datalen) >= next){
                        if ((counter*100/datalen) < 100) {
                                System.out.print(" "+Integer.toString(next)+"%");
            System.out.flush();
                        }
        else {
                System.out.println(" 100%");
        }
        next++;
                }
```

```
                    ID = indexF.readLine();
                    fileAddress1 = indexF.readLine();
                    fileAddress2 = indexF.readLine();
                    output = "";
                }

                out.close();
        }

        //from pitt.search.semanticvectors.VectorSearcher
        public static float getScore(String[] stringArray1, String[] stringArray2,
                VectorStore vecReader, boolean info) throws Exception{
                float score;
                float[] vector1 = getVector(stringArray1, vecReader, info);
                float[] vector2 = getVector(stringArray2, vecReader, info);

                if (!VectorUtils.isZeroVector(vector1)) {
                if (!VectorUtils.isZeroVector(vector2)) {
                                score = VectorUtils.scalarProduct(vector1, vector2);
                                //more mathematical operation in pitt.saerch.semanticvectors.VectorSearcher
                        } else throw new Exception("The vector for stringArray2 is a ZeroVector.");
                } else throw new Exception("The vector for stringArray1 is a ZeroVector");

                return score;
        }

        //from pitt.search.semanticvectors.CompoundVectorBuilder
        public static float[] getVector (String[] queryTerms, VectorStore vecReader, boolean info){
            int dimension = 200;
                float[] queryVec = new float[dimension];
            float[] tmpVec = new float[dimension];
                StringBuffer queryTermsUsed = new StringBuffer();

            for (int i = 0; i < dimension; ++i) {
                queryVec[i] = 0;
              }

            for (int j = 0; j < queryTerms.length; ++j) {
                tmpVec = vecReader.getVector(queryTerms[j]);


                if (tmpVec != null) {
                    for (int i = 0; i < dimension; ++i) {
                      queryVec[i] += tmpVec[i];
                    }
                    if(info) queryTermsUsed.append (queryTerms[j] + " ");
                  } else {
                                        //if(info)System.err.println("No vector for " + queryTerms[j]);
                                }
            }
            if(info) System.out.println("Words used to build the vector : " + queryTermsUsed);

            return VectorUtils.getNormalizedVector(queryVec);
        }

}
```

B.6 evaluatecorpus.py

```
#!/usr/local/bin/python
"""A script for running the project experiment.

Written by Edward Grefenstette, University of Oxford, 2009.
email: <edward.grefenstette@balliol.ox.ac.uk>
Submitted towards completion of MSc Computer Science.
"""

from __future__ import division
import sys, os
import cPickle as pickle
from corpusevaluation import *
import hashlib

def main():
    corpusfile = open(r"/Users/Edward/Dropbox/MSc CS/MSc Project/" \
                      "corpus/preppedcorp.pkl","r")

    print "Loading pre-processed evaluation data."
    eval_data = pickle.load(corpusfile)
    print "Done reloading evaluation data."

    # To check compatibility during running backup
    print "Calculating corpus signature for running backup."
    corpusfile.seek(0)
    signature = hashlib.md5(corpusfile.read()).digest()
    corpusfile.close()
    print "Done calculating signature."

    results = getmetricresults(eval_data,signature)
    publishresults(results)

def getmetricresults(eval_data,signature):

    BKUPpath = r"/Users/Edward/Dropbox/MSc CS/MSc Project/" \
                      "analysis/runningbackup.pkl"

    metrics = {"Wordcount": wordcounteval, "Character Count": charcounteval,
               "Levenshtein Edit Distance": leveneval,
               "Jaro-Winkler Distance": jaroeval,
               "Ratio Similarity": ratioeval, "Jaccard Distance": jaccardeval,
               "Masi Distance": masieval, "Random Evaluation": randomeval,
               "WN Path Similarity": pathsimeval,
               "WN Wu-Palmer Similarity": WUPeval,
               "WN Lin Similarity": LINeval,
               "Semantic Vector Similarity": SemVecteval,
               "BLEU Similarity": BLEUeval
               }

    metricnames = {}
    for key in metrics:
        metricnames[metrics[key]]=key

    evalTODO = metrics.keys()
```

```
metricresults = {}

print "Checking for existing backups."
if os.path.exists(BKUPpath):
    print "Backup file found! Validating..."
    loadbackup = True
    BKUPf = open(BKUPpath)
    BKsignature,BKevalTODO,BKevalDONE,BKmetricresults = pickle.load(BKUPf)
    if not (BKsignature == signature):
        print "Corpus signature does not match."
        loadbackup = False
    BKevalDONEtgt = set(metrics.keys()).difference(set(BKevalTODO))
    if not (BKevalDONEtgt == BKevalDONE):
        print "Metric set does not match."
        loadbackup = False
    if loadbackup:
        print "Reloading results for metrics:"
        for metricname in BKevalDONE:
            print "\t",metricname
        evalTODO = BKevalTODO
        metricresults = BKmetricresults
    BKUPf.close()
    os.remove(BKUPpath)

runningBKUP = open(BKUPpath,"w")

print "Beginning evaluation."
try:
    while len(evalTODO) > 0:
        key = evalTODO[0]
        metric = metrics[key]
        name = metricnames[metric]
        print "==Evaluating "+name+"=="
        scorer = metric(eval_data=eval_data)
        metricresults[name] = scorer.scores
        del(evalTODO[0])
except:
    print "Something went wrong while executing metric:", evalTODO[0]
    evalDONE = set(metrics.keys()).difference(set(evalTODO))
    if len(evalDONE) is 0:
        print "No results to record..."
        sys.exit()
    pickle.dump((signature,evalTODO,evalDONE,metricresults), runningBKUP)
    runningBKUP.close()

    print "Progress saved. Completed metrics:"
    for metricname in evalDONE:
        print "\t",metricname
    print "Left to be evaluated:"
    for metricname in evalTODO:
        print "\t",metricname
    sys.exit()

runningBKUP.close()
os.rename(BKUPpath, BKUPpath+".final")

print "Done evaluating metrics."
```

```
    return metricresults

def publishresults(results):
    outputfileloc = r"/Users/Edward/Dropbox/MSc CS/MSc Project/" \
                    "analysis/results.pkl"
    outputfile = open(outputfileloc,"w")
    pickle.dump(results, outputfile)
    outputfile.close()

if __name__ == "__main__":
    try:
        print "Importing psyco..."
        import psyco
        psyco.full()
    except ImportError:
        pass
    main()
```

B.7 analyser.py

```
#!/usr/local/bin/python
"""A script for analysing the experiment results.

Written by Edward Grefenstette, University of Oxford, 2009.
email: <edward.grefenstette@balliol.ox.ac.uk>
Submitted towards completion of MSc Computer Science.
"""

from __future__ import division
import sys, os
from optparse import OptionParser
import cPickle as pickle
import re
import numpy as np
import pylab

def main():
    if not options.quiet: print "Loading results for analysis."
    try:
        resultsFile = open(args[0])
        results = pickle.load(resultsFile)
        resultsFile.close()
    except:
        sys.stderr.write("Could not load results from file "+args[0] \
                         +". Quitting.\n")
        sys.exit()
    if not options.quiet: print "Done loading results."

    # get categories:
    cats = {}
    matchcats = {}
    mixedcats = {}
```

```
for tag in results[results.keys()[0]]:
    parts = re.search(r"([a-zA-Z]+)-*([a-zA-Z]+)*", tag)
    cat = parts.group(1)
    subcat = parts.group(2)
    try: cats[cat].append(tag)
    except: cats[cat] = [tag]

    if subcat is not None:
        try: cats[cat+" ["+subcat+"]"].append(tag)
        except:
            try: cats[cat+" ["+subcat+"]"] = [tag]
            except: cats[cat+" ["+subcat+"]"] = [tag]

    if tag[-1] == 'm':
        try: mixedcats[cat].append(tag)
        except: mixedcats[cat] = [tag]

        if subcat is not None:
            try: mixedcats[cat+" ["+subcat+"]"].append(tag)
            except:
                try: mixedcats[cat+" ["+subcat+"]"] = [tag]
                except: mixedcats[cat+" ["+subcat+"]"] = [tag]

    else:
        try: matchcats[cat].append(tag)
        except: matchcats[cat] = [tag]

        if subcat is not None:
            try: matchcats[cat+" ["+subcat+"]"].append(tag)
            except:
                try: matchcats[cat+" ["+subcat+"]"] = [tag]
                except: matchcats[cat+" ["+subcat+"]"] = [tag]

# splice up scores per metric
catscores = getcatscores(cats, results)
matchcatscores = getcatscores(matchcats, results)
mixedcatscores = getcatscores(mixedcats, results)

# calculate metric averages per category (overall)
metricAVGs = getmetricAVGs(cats, catscores, results)
mixedmetricAVGs = getmetricAVGs(mixedcats, mixedcatscores, results)
matchmetricAVGs = getmetricAVGs(matchcats,matchcatscores,results)

# publish metric averages
publishAVGs(metricAVGs,"Overall Averages",True)
publishAVGs(matchmetricAVGs,"Averages for Matched Doc Pairs")
publishAVGs(mixedmetricAVGs,"Averages for Mixed Doc Pairs")

# publish results distributions
if not options.textonly:
    publishDists(catscores,cats,"Overall")
    publishDists(matchcatscores,matchcats,"Matched")
    publishDists(mixedcatscores,mixedcats,"Mixed")

metriclist = [metric for metric in results]
metricpairs = []
for i in range(0,len(metriclist)-1):
```

```
        for j in range(i+1,len(metriclist)):
            metricpairs.append((metriclist[i],metriclist[j]))

    metdiffs = getmetricdiffs(metricpairs, results)

    publishDiffs(metdiffs,cats,"Overall Metric Differences",True)
    publishDiffs(metdiffs,matchcats,"Metric Differences for Matched Doc Pairs")
    publishDiffs(metdiffs,mixedcats,"Metric Differences for Mixed Doc Pairs")

    if not options.textonly:
        publishDiffDists(metdiffs,cats,"Overall")
        publishDiffDists(metdiffs,matchcats,"Matched")
        publishDiffDists(metdiffs,mixedcats,"Mixed")

def getcatscores(cats,results):
    catscores = {}
    for metric in results:
        catscores[metric] = {}
        for cat in cats:
            catscores[metric][cat] = {}
            for tag in cats[cat]:
                catscores[metric][cat][tag] = results[metric][tag]
    return catscores

def getmetricAVGs(cats,catscores,results):
    metricAVGs = {}
    for cat in cats:
        metricAVGs[cat] = {}
        for metric in results:
            scores = []
            for tag in catscores[metric][cat]:
                scores.append(catscores[metric][cat][tag])
                metricAVGs[cat][metric] = sum(scores)/len(scores)
    return metricAVGs

def publishAVGs(AVGs,title,removeOrig=False):
    rankdict = {}
    for cat in AVGs:
        rankdict[cat] = {}
        for metric in AVGs[cat]:
            rankdict[cat][AVGs[cat][metric]] = metric

    ranks = {}
    for cat in rankdict:
        ranks[cat] = []
        sorted = rankdict[cat].keys()
        sorted.sort() # ascending sort
        sorted.reverse() # reverse to have high scores first
        for key in sorted:
            ranks[cat].append((rankdict[cat][key],key))

    sortedcats = ranks.keys()
    sortedcats.sort()

    if options.textonly:
        print "=====",title,"====="
        for cat in sortedcats:
```

```
            print "=== Category", cat, "==="
            for i in range(0,len(ranks[cat])):
                print "\t"+str(i+1)+": %-20s" % str(ranks[cat][i][0])+"\t" + \
                                            "%6.2f%%" % (ranks[cat][i][1] * 100)
            print "" # blank line between categories

    else:
        content = "===== "+title+" =====\n"
        for cat in sortedcats:
            content += "=== Category "+cat+" ===\n"
            for i in range(0,len(ranks[cat])):
                content += "\t"+str(i+1)+": %-30s" % str(ranks[cat][i][0])+ \
                                    "\t"+"%6.2f%%" % (ranks[cat][i][1]*100)+"\n"
            content += "\n"
        try:
            if not options.quiet: print "Printing "+title+" to rankings.txt."
            savepath = os.path.abspath(args[1]) + "/rankings.txt"
            if removeOrig:
                if os.path.exists(savepath): os.remove(savepath)
            outputfile = open(savepath,"a")
            outputfile.write(content)
            outputfile.close()
        except:
            sys.stderr.write("Couldn't write to folder"+args[1]+". Quitting.\n")
            sys.exit()

def publishDists(catscores, cats, subfolder):
    fpath = os.path.abspath(args[1])
    for cat in cats:
        catpath = fpath+"/GSdists/"+cat
        if not os.path.exists(catpath):
            os.makedirs(catpath)
        outputpath = catpath+"/"+subfolder
        if not os.path.exists(outputpath):
            os.mkdir(outputpath)
        for metric in catscores:
            savepath = outputpath+"/"+metric+" "+cat+".pdf"
            if os.path.exists(savepath): os.remove(savepath)
            data = [catscores[metric][cat][tag]
                    for tag in catscores[metric][cat]]
            pylab.clf()
            pylab.hist(data,bins=99)
            pylab.title(cat+" Corpus ("+subfolder+")\n" + \
                        "Score difference distribution for metric: "+metric)
            pylab.xlabel("Closeness to gold standard\n[Higher is better]")
            pylab.ylabel("Number of occurrences")
            pylab.xlim((0.0,1.0))
            pylab.savefig(savepath,format="pdf")
            pylab.close()

def getmetricdiffs(metricpairs,results):
    metricdiffs = {}
    for pair in metricpairs:
        metricdiffs[pair] = {}
        met1, met2 = pair
        for ID in results[met1]:
            metricdiffs[pair][ID] = abs(results[met1][ID]-results[met2][ID])
```

```
    return metricdiffs

def publishDiffs(metdiffs,cats,title,removeOrig=False):
    rankdict = {}

    diffAVGs = {}
    for cat in cats:
        diffAVGs[cat] = {}
        for pair in metdiffs:
            AVG = sum([metdiffs[pair][ID] for ID in cats[cat]])/len(cats[cat])
            diffAVGs[cat][pair] = AVG

    for cat in cats:
        rankdict[cat] = {}
        for pair in metdiffs:
            pairname = pair[0] + " vs. " + pair[1]
            rankdict[cat][diffAVGs[cat][pair]] = pairname

    ranks = {}
    for cat in rankdict:
        ranks[cat] = []
        sorted = rankdict[cat].keys()
        sorted.sort() # ascending sort
        sorted.reverse() # reverse to have high scores first
        for key in sorted:
            ranks[cat].append((rankdict[cat][key],key))

    sortedcats = ranks.keys()
    sortedcats.sort()

    if options.textonly:
        print "=====",title,"====="
        for cat in sortedcats:
            print "=== Category", cat, "==="
            for i in range(0,len(ranks[cat])):
                print "\t"+str(i+1)+": %-70s" % str(ranks[cat][i][0])+"\t" + \
                                            "%6.2f%%" % (ranks[cat][i][1] * 100)
            print "" # blank line between categories

    else:
        content = "===== "+title+" =====\n"
        for cat in sortedcats:
            content += "=== Category "+cat+" ===\n"
            for i in range(0,len(ranks[cat])):
                content += "\t"+str(i+1)+": %-70s" % str(ranks[cat][i][0])+ \
                                "\t"+"%6.2f%%" % (ranks[cat][i][1]*100)+"\n"
            content += "\n"
        try:
            if not options.quiet: print "Printing "+title+" to metricdiffs.txt."
            savepath = os.path.abspath(args[1]) + "/metricdiffs.txt"
            if removeOrig:
                if os.path.exists(savepath): os.remove(savepath)
            outputfile = open(savepath,"a")
            outputfile.write(content)
            outputfile.close()
        except:
            sys.stderr.write("Couldn't write to folder"+args[1]+". Quitting.\n")
```

```
            sys.exit()


def publishDiffDists(metdiffs, cats, subfolder):
    fpath = os.path.abspath(args[1])
    for cat in cats:
        catpath = fpath+"/Diffdists/"+cat
        if not os.path.exists(catpath):
            os.makedirs(catpath)
        outputpath = catpath+"/"+subfolder
        if not os.path.exists(outputpath):
            os.mkdir(outputpath)
        for pair in metdiffs:
            fname = pair[0] + " vs " + pair[1]
            savepath = outputpath+"/"+fname+" "+cat+".pdf"
            if os.path.exists(savepath): os.remove(savepath)
            data = [metdiffs[pair][ID] for ID in cats[cat]]
            pylab.clf()
            pylab.hist(data,bins=99)
            pylab.title(cat+" Corpus ("+subfolder+")\n" + \
                        "Metric difference for: "+fname)
            pylab.xlabel("Score diversion\n")
            pylab.ylabel("Number of occurrences")
            pylab.xlim((0.0,1.0))
            pylab.savefig(savepath,format="pdf")
            pylab.close()


if __name__ == "__main__":
    usage = "%prog [-t] DATA OUTPUTFOLDER"
    description = ""
    parser = OptionParser(usage = usage, description = description)
    parser.add_option("-t", "--text-only", dest="textonly",
                      default = False, help="Write text-only results to STDIN",
                      action = "store_true")
    parser.add_option("-q", "--quiet", dest="quiet",
                      default = False, help="Suppress progress outputs",
                      action = "store_true")
    (options, args) = parser.parse_args()

    if (len(args) < 1) or ((not options.textonly) and len(args)<2):
        parser.print_help()
        sys.exit()

    if not options.textonly:
        if not os.path.exists(os.path.abspath(args[1])):
            sys.stderr.write("Could not find folder for saving analysis.\n")
            sys.exit(s)

    main()
```